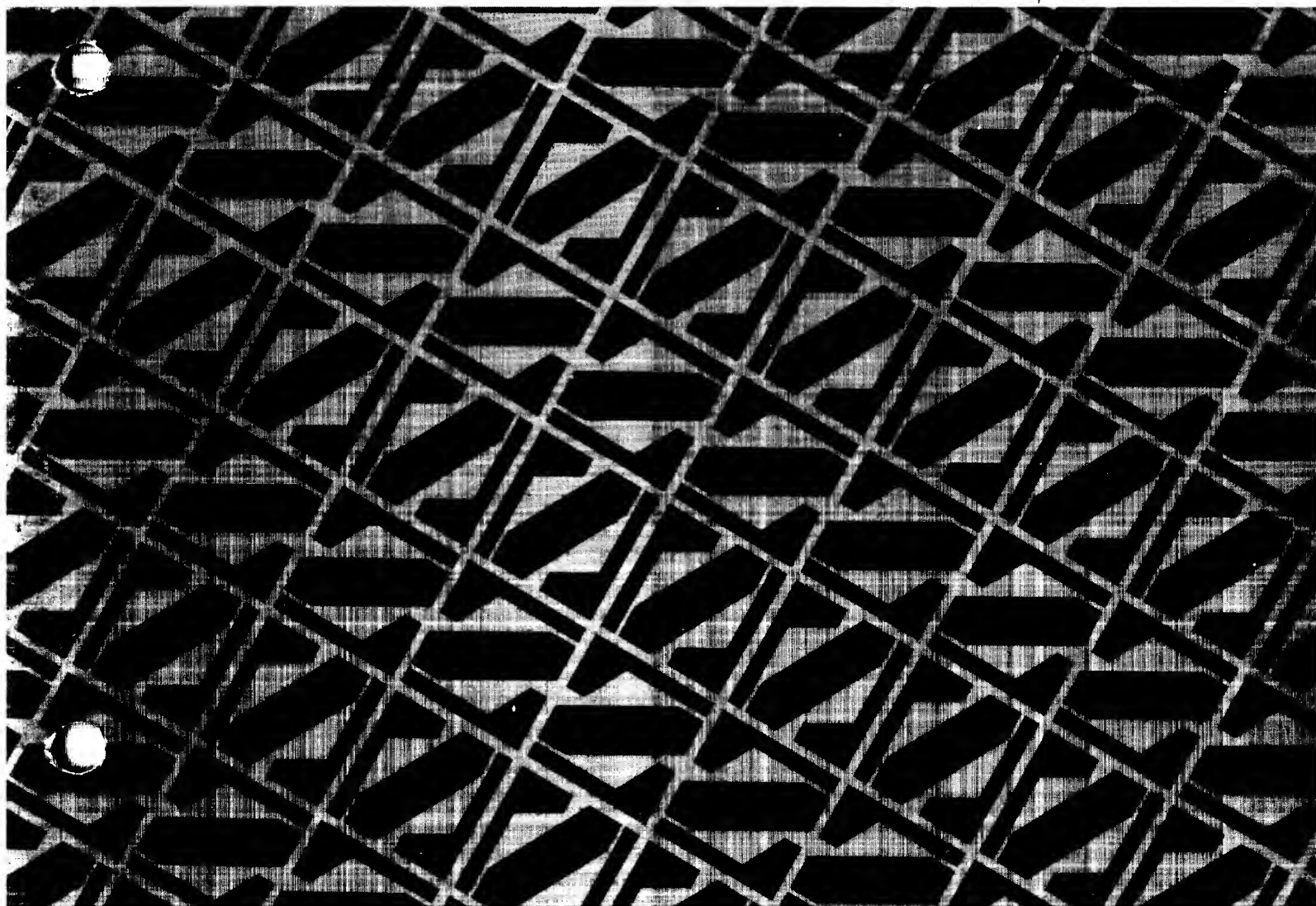


National Semiconductor
Order No. IMP-16S/102YB
Pub. No. 4200002B

IMP-16

Programming and Assembler Manual



Order Number IMP-16S/102YB
Publication Number 4200002B

Integrated MicroProcessor-16

IMP-16

PROGRAMMING AND ASSEMBLER MANUAL

November 1973

© National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051

PREFACE

The IMP-16 Programming and Assembler Manual provides tutorial and reference information for devising user application programs. The manual is written for the benefit of both engineers and programmers for IMP-16 programming indoctrination. Information pertaining to the IMP-16 microprocessor and microcomputer equipment is not provided in this manual, nor is other programming information listed in appendix J (References) provided; these references should be consulted for the appropriate information.

This issue supersedes the preceding issues of 4200002A (formerly titled "IMP-16 Assembler Manual").

Changes to this manual will be provided in the form of change notices or change pages to be added to or to replace pages. Such changes will be reflected in a List of Effective Pages, to be provided with each change.

The material in this manual is for information purposes only and is subject to change without notice.

It is suggested that the reader thoroughly review the tables of contents, illustrations, and tables to familiarize himself with an overview of the organization of the manual before reading the contents. By so doing, the reader may then be prepared to appreciate the extent-of-coverage; such an appreciation shall likely be useful during the initial reading of this manual.

Copies of this publication and other National Semiconductor publications may be obtained from the sales offices listed on the back cover.

CONTENTS

Chapter		Page
1	INTRODUCTION	1-1
2	INTRODUCTION TO ASSEMBLER PROGRAMMING	2-1
	2.1 THE ASSEMBLER LANGUAGE	2-1
	2.2 THE ASSEMBLER PROGRAM	2-2
	2.3 ASSEMBLER PROGRAMMING AIDS	2-6
	2.3.1 Data Representation	2-6
	2.3.2 Binary Fixed-Point Format	2-6
	2.3.3 Memory Addressing	2-6
	2.3.4 Relocatability	2-7
	2.3.5 Linking	2-7
3	OVERVIEW OF CPU	3-1
	3.1 OPERATIONAL FEATURES	3-1
	3.2 ARITHMETIC AND LOGIC UNITS REFERENCED BY IMP-16 ASSEMBLER LANGUAGE AND PROGRAM	3-1
	3.2.1 Last-In/First-Out Stack (LIFOS)	3-1
	3.2.2 Status Register	3-1
	3.2.3 Program Counter (PC)	3-2
	3.2.4 Memory Data Register (MDR) and Memory Address Register (MAR).	3-2
	3.2.5 Working Registers	3-2
4	IMP-16 ASSEMBLER LANGUAGE	4-1
	4.1 IMP-16 ASSEMBLER CODING CONVENTIONS	4-1
	4.1.1 Label Field	4-1
	4.1.2 Operation Field	4-1
	4.1.3 Operand Field	4-1
	4.1.4 Comment Field	4-1
	4.1.5 Identification Sequence Field	4-3
	4.1.6 General Statements	4-3
	4.1.7 Example Statement	4-3
	4.1.8 Character Set	4-3
	4.2 IMP-16 ASSEMBLER LANGUAGE STRUCTURE	4-3
	4.2.1 Label Entry	4-4
	4.2.2 Operation Mnemonic Entry	4-4
	4.2.3 Operand Entry	4-4
	4.3 DATA REPRESENTATION	4-9

CONTENTS (Continued)

Chapter		Page
5	ADDRESSING	5-1
5.1	REGISTERS	5-1
5.2	MEMORY	5-1
5.3	LOCATION COUNTER	5-1
5.4	METHODS OF ADDRESSING	5-2
5.4.1	Immediate Addressing	5-2
5.4.2	Direct Addressing	5-2
5.4.3	Indirect Addressing	5-3
5.4.4	Double-Word Addressing	5-3
6	IMP-16 ASSEMBLER PROGRAMS	6-1
6.1	DEFINITION	6-1
6.2	PROGRAM RELOCATION	6-1
6.3	INPUT AND OUTPUT	6-3
6.3.1	Source File (Input)	6-3
6.3.2	Program Listing File (Output)	6-3
6.3.3	Relocatable Load Module (Output)	6-4
6.4	TYPES OF RLM RECORDS	6-4
6.4.1	Title Record	6-4
6.4.2	Symbol Records	6-4
6.4.3	Data Record	6-8
6.4.4	End Record	6-9
6.5	LOADING OBJECT PROGRAM INTO IMP-16	6-9
6.5.1	Bootstrap Loaders	6-10
6.5.2	Absolute Loaders	6-10
6.5.3	Linking Loader	6-12
6.5.4	Reformatting RLM Output on Other Media or in Other Formats	6-12
7	INSTRUCTION STATEMENTS	7-1
7.1	INTRODUCTION	7-1
7.2	LOAD AND STORE INSTRUCTIONS (Basic Set)	7-3
7.2.1	Load Direct (LD)	7-3
7.2.2	Load Indirect (LD)	7-4
7.2.3	Store Direct (ST)	7-4
7.2.4	Store Indirect (ST)	7-4
7.3	BYTE INSTRUCTIONS (Extended Set)	7-5
7.3.1	Load Byte (LDB)	7-5
7.3.2	Load Left Byte (LLB)	7-5
7.3.3	Load Right Byte (LRB)	7-5
7.3.4	Store Byte (STB)	7-6
7.3.5	Store Left Byte (SLB)	7-6
7.3.6	Store Right Byte (SRB)	7-6

CONTENTS (Continued)

Chapter		Page
7	INSTRUCTION STATEMENTS (Continued)	
7.4	SINGLE-PRECISION ARITHMETIC INSTRUCTIONS (Basic and Extended Sets)	7-6
7.4.1	Add (ADD).	7-7
7.4.2	Subtract (SUB)	7-7
7.4.3	Multiply (MPY)	7-7
7.4.4	Divide (DIV)	7-8
7.5	DOUBLE-PRECISION ARITHMETIC INSTRUCTIONS (Extended Set)	7-8
7.5.1	Double-Precision Add (DADD).	7-9
7.5.2	Double-Precision Subtract (DSUB)	7-9
7.6	LOGICAL INSTRUCTIONS (Basic Set)	7-10
7.6.1	Logical AND (AND)	7-10
7.6.2	Logical OR (OR)	7-10
7.7	REGISTER INSTRUCTIONS (Basic Set)	7-11
7.7.1	Push onto Stack (PUSH).	7-11
7.7.2	Pull from Stack (PULL).	7-12
7.7.3	Exchange Register and Stack (XCHRS)	7-12
7.7.4	Load Immediate (LI).	7-12
7.7.5	Add Immediate, Skip if Zero (AISZ)	7-13
7.7.6	Complement and Add Immediate (CAI)	7-13
7.7.7	Register Add (RADD)	7-14
7.7.8	Register Exchange (RXCH).	7-14
7.7.9	Register Copy (RCPY)	7-14
7.7.10	Register Exclusive OR (RXOR)	7-15
7.7.11	Register AND (RAND)	7-15
7.8	BIT AND STATUS FLAG INSTRUCTIONS (Extended and Basic Sets)	7-15
7.8.1	Push Status Flags onto Stack (PUSHF)	7-17
7.8.2	Pull Status Flags from Stack (PULLF)	7-17
7.8.3	Set Status Flag (SETST).	7-17
7.8.4	Clear Status Flag (CLRST).	7-18
7.8.5	Set Bit (SETBIT).	7-18
7.8.6	Clear Bit (CLRBIT)	7-18
7.8.7	Complement Bit (CMPBIT).	7-18
7.9	TRANSFER-OF-CONTROL INSTRUCTIONS (Basic and Extended Sets)	7-19
7.9.1	Jump Direct (JMP)	7-19
7.9.2	Jump Indirect (JMP)	7-20
7.9.3	Jump Through Pointer (JMPP).	7-20
7.9.4	Jump to Subroutine Direct (JSR)	7-20
7.9.5	Jump to Subroutine Indirect (JSR).	7-21
7.9.6	Jump to Subroutine Implied (JSRI).	7-21
7.9.7	Jump to Subroutine Through Point (JSRP)	7-21
7.9.8	Return from Subroutine (RTS)	7-22
7.9.9	Return from Interrupt (RTI)	7-22
7.9.10	Branch on Condition (BOC).	7-22

CONTENTS (Continued)

Chapter		Page
7	INSTRUCTION STATEMENTS (Continued)	
7.10	SKIP INSTRUCTIONS (Basic and Extended Sets)	7-23
7.10.1	Increment and Skip if Zero (ISZ)	7-24
7.10.2	Decrement and Skip if Zero (DSZ)	7-24
7.10.3	Skip if Greater (SKG)	7-24
7.10.4	Skip if Not Equal (SKNE)	7-25
7.10.5	Skip if AND is Zero (SKAZ)	7-25
7.10.6	Skip if Status Flag True (SKSTF).	7-25
7.10.7	Skip if Bit True (SKBIT)	7-26
7.11	SHIFT INSTRUCTIONS (Basic Set)	7-26
7.11.1	Shift Left (SHL)	7-26
7.11.2	Shift Right (SHR)	7-27
7.11.3	Rotate Left (ROL)	7-27
7.11.4	Rotate Right (ROR)	7-28
7.12	INTERRUPT HANDLING INSTRUCTIONS (Extended Set)	7-28
7.12.1	Interrupt Scan (ISCAN).	7-28
7.12.2	Jump to Level 0 Interrupt, Indirect (JINT).	7-29
7.13	INPUT/OUTPUT, HALT, AND CONTROL FLAG INSTRUCTIONS (Basic Set)	7-29
7.13.1	Register IN (RIN)	7-30
7.13.2	Register Out (ROUT)	7-30
7.13.3	Halt (HALT)	7-30
7.13.4	Set Flag (SFLG).	7-31
7.13.5	Pulse Flag (PFLG).	7-31
8	ASSIGNMENT STATEMENT	8-1
9	DIRECTIVE STATEMENTS	9-1
9.1	TITLE DIRECTIVE (.TITLE)	9-2
9.2	PROGRAM SECTION DIRECTIVES (.ASECT, .BSECT, .TSECT)	9-2
9.3	END DIRECTIVE (.END)	9-2
9.4	LIST DIRECTIVE (.LIST)	9-3
9.5	SPACE DIRECTIVE (.SPACE).	9-3
9.6	PAGE DIRECTIVE (.PAGE)	9-3
9.7	WORD DIRECTIVE (.WORD)	9-3
9.8	ASCII DIRECTIVE (.ASCII)	9-4
9.9	GLOBAL DIRECTIVE (.GLOBL)	9-4
9.10	LOCAL DIRECTIVE (.LOCAL)	9-4
9.11	CONDITIONAL ASSEMBLY DIRECTIVES (.IF, .ELSE, .ENDIF)	9-5
9.12	FORM DIRECTIVE (.FORM)	9-6
9.13	EXTENDED INSTRUCTION DIRECTIVE (.EXTD)	9-7

CONTENTS (Continued)

Chapter		Page
10	IMP-16 RESIDENT ASSEMBLER OPERATING PROCEDURE	10-1
	10.1 ASSEMBLER LOADING PROCEDURE	10-1
	10.2 ASSEMBLING A PROGRAM	10-2
	10.3 CARD READER INPUT	10-2
	10.4 PAPER TAPE INPUT	10-2
	10.5 KEYBOARD INPUT	10-3
	10.6 KEYBOARD/PAPER TAPE SPECIAL EDITING CHARACTERS	10-3
	10.7 OBJECT LISTING	10-3
	10.8 RELOCATABLE LOAD MODULE (RLM)	10-3
A	APPENDIX — CHARACTER SETS	A-1
B	APPENDIX — STATUS BITS IN ARITHMETIC OPERATIONS.	B-1
	B.1 ARITHMETIC WITH UNSIGNED DATA WORDS	B-1
	B.2 MULTIPLICATION AND DIVISION	B-3
C	APPENDIX — INPUT/OUTPUT PROGRAMMING	C-1
	C.1 PROGRAMMED INPUT/OUTPUT AND INTERRUPT INPUT/OUTPUT	C-1
	C.1.1 Programmed Input/Output	C-1
	C.1.2 Interrupt Input/Output	C-1
	C.1.3 Stack Full Interrupt	C-2
	C.2 INPUT/OUTPUT SYSTEM ORGANIZATION	C-4
	C.2.1 Generalized Call to Input/Output	C-4
	C.2.2 Device Drivers	C-5
D	APPENDIX — PROGRAMMERS CHECKLIST	D-1
E	APPENDIX — FOLD16 - IMP-16 FORTRAN OBJECT LOADER PROGRAM DESCRIPTION	E-1
	E.1 GENERAL USAGE INFORMATION	E-1
	E.2 IMPLEMENTATION	E-2
	E.3 CALLING CONDITIONS	E-2
	E.4 RETURNING CONDITIONS	E-3
	E.5 DESCRIPTION OF OPERATION	E-3
	E.6 ENTRY NAME: UNPACK	E-3
	E.6.1 Purpose	E-3
	E.6.2 Calling Conditions	E-3
	E.6.3 Returning Conditions	E-3

CONTENTS (Continued)

Chapter		Page
F	APPENDIX — PROGRAM DIAGNOSTIC MESSAGES	F-1
	F.1 INTRODUCTION.	F-1
	F.2 RESIDENT ASSEMBLER ERROR MESSAGES	F-1
	F.3 CROSS ASSEMBLER ERROR MESSAGES	F-2
	F.4 OTHER ERROR CONDITIONS	F-4
G	APPENDIX — DIRECTIVE STATEMENTS.	G-1
H	APPENDIX — INDEX OF INSTRUCTION STATEMENTS	H-1
I	APPENDIX — CONVERSION TABLES	I-1
J	APPENDIX — REFERENCES	J-1

ILLUSTRATIONS

Figure		Page
2-1	Sample Source Program	2-3
2-2	Example of IMP-16 Cross Assembler Listing	2-4
2-3	Programming Process	2-5
3-1	Arithmetic and Logic Units Referenced by IMP-16 Assembler	3-2
4-1	Sample Coding Form	4-2
6-1	Memory Map	6-2
6-2	RLM File and General Record Formats	6-5
6-3	Title Record Format	6-6
6-4	Symbol Record Format	6-7
6-5	Data Record Format	6-8
6-6	End Record Format	6-9
6-7	Operational Sequence for Preloading and Generating Memory Image Deck for Loading by CRBOOT	6-11
6-8	Operational Sequence for Preparation of Input for ABSCR or GENLDR	6-13
7-1	Configuration of Status Register	7-16
9-1	Example of Conditional Assembly Directives	9-5
10-1	Sample Listing of Resident Assembler	10-4
F-1	Resident Assembler Error Detection, Listing Output	F-1
F-2	Cross Assembler Error Detection, Listing Output	F-2

TABLES

Number		Page
4-1	Expression Classification Table	4-7
5-1	Address Operands.	5-4
5-2	Assembler Execution of Direct Base Page or Program-Counter-Relative Address. . .	5-6
5-3	Assembler Execution of Indirect Base Page or Program-Counter-Relative Address . .	5-7
5-4	Assembler Execution of Direct Indexed Address	5-8
5-5	Assembler Execution of Indirect Indexed Address	5-9
5-6	Assembler Execution of Direct or Indirect Indexed Address Without Displacement Value .	5-9
7-1	Notations and Symbols Used in Operational Descriptions	7-2
7-2	Definitions of IMP-16C/L Flags	7-16
7-3	Branch On Condition Codes	7-23
7-4	Control Flags	7-31
9-1	Summary of Assembler Directives	9-1
A-1	ANSI Character Set in Hexadecimal Representation	A-1
A-2	Legend for Nonprintable Characters	A-2
G-1	Index of Directive Statements	G-1
H-1	Index of Basic Instruction Statements	H-1
H-2	Index of Extended Instruction Statements	H-3
H-3	Definitions of Field Designators	H-4
I-1	Positive Powers of Two	I-1
I-2	Negative Powers of Two	I-2
I-3	Hexadecimal and Decimal Integer Conversion Table	I-3
I-4	Hexadecimal and Decimal Fraction Conversion Table	I-4
I-5	Integer Conversion Table	I-4

Chapter 1

INTRODUCTION

This manual describes the IMP-16 Assembler Language and Programs. It is intended as a reference manual to assist the IMP-16 user in developing software.

The IMP-16 is a programmable 16-bit parallel microprocessor. The CPU is configured around the National Semiconductor General-Purpose Controller/Processor MOS/LSI devices. The MOS/LSI devices consist of one or two CROMs (Control Read Only Memory), and four RALUs (Register and Arithmetic Logic Units). Each RALU handles 4 bits, and a 16-bit unit is formed by connecting four RALUs in parallel. Up to 65,536 words of semiconductor memory are supported.

The user has the alternative to select between two IMP-16 Assemblers: the IMP-16 Resident Assembler and the IMP-16 Cross Assembler. Both assemblers are completely compatible in the programs they assemble and vary only in their operating environments. In this manual, references to "the assembler program" will be concerned with information common to both assemblers.

The IMP-16 Resident Assembler runs on an IMP-16 computer with a minimum of 4K words of memory and a Teletype. The IMP-16 Resident Assembler accepts free format source statements from either the keyboard, paper tape, or a card reader and produces an unlinked Relocatable Load Module on paper tape and an object listing on the Teletype printer. The IMP-16 Resident Assembler requires three passes over the source program; however, if the object listing or Relocatable Load Module is to be suppressed, only two passes are required.

The information in this publication is based on the assumption that the reader is familiar with electronic data processing hardware functions, but may or may not have assembler programming experience. Introductory material is provided for the user with little or no assembler programming experience. Chapter 2 contains a brief introduction to programming. Chapter 4 discusses the IMP-16 Assembler language in detail, and, when necessary, the logical concepts upon which machine instructions are based are defined in chapter 7. It is recommended that the user with no programming experience study the introductory sections and read through the entire manual before trying to analyze some of the more-complex features.

All IMP-16 users should read chapter 5 on addressing carefully. The IMP-16 Cross Assembler Program is discussed in chapter 6, and the machine and assembler statements are discussed in chapters 7 through 9. The IMP-16 Resident Assembler is discussed in chapter 10.

Chapter 2

INTRODUCTION TO ASSEMBLER PROGRAMMING

This chapter describes basic programming concepts for the user with no programming background. The following discussion is based on the assumption that the reader is familiar with electronic data processing and the binary and hexadecimal numbering systems.

A program is a list of instructions in a specific sequence defined by the programmer to operate on data. An instruction is a statement that contains two basic parts: an operation code defining the operation to be performed and one or more operands defining the location of the data or specifying a device to be used.

The sequence of instructions in the program follow performs the following functions:

Establishes working areas (areas to which data is moved for manipulation) in storage.

Specifies constants (values used in arithmetic calculations, symbols used to set switches, and so forth).

Specifies the appropriate operations to move data, perform appropriate tests and calculations, handle exceptional conditions, and arrange data in appropriate output formats.

Many programmers find a flowchart assists in coding instruction statements. A process flowchart contains all of the information that a programmer needs to write a usable program. Usually each step to be coded in a statement line is represented by one or more symbols. Some of the symbols represent data manipulation activities. Others represent operations that are required by the processor. Housekeeping activities such as setting counters or clearing output areas are typical examples of processor operations.

2.1 THE ASSEMBLER LANGUAGE

An assembler language is a machine-oriented symbolic programming language that allows the programmer to specify operations and operands with symbolic notations instead of binary notation. The programmer specifies alphabetic or alphanumeric symbols in place of memory addresses for data and instructions. In addition, the assembler language provides mnemonic operations codes.

The following example contrasts writing one instruction in binary and in assembler language.

Assume the programmer wants to terminate a sequence of instructions and start another. In the following statement, the programmer codes the binary numbers in hexadecimal, one digit per column.

Column	1	2	3	4
	2	1	F	5

The assembler language permits the programmer to use symbolic and/or decimal notation. The same instruction in assembler language looks like this:

JMP LOOP

In the hexadecimal coding, the programmer has the responsibility of counting address locations and determining if the jump (in this example) would be best made through the base page, or performed as a displacement from an indexed value, or a displacement from the present position in memory. If the address to which the jump is to be made is labeled LOOP:, the assembler takes care of all housekeeping.

The assembler program keeps track of memory addresses for symbolic terms and substitutes these addresses in place of the symbolic name in the object code instruction that is executed.

In summary, a symbolic language gives the user several important advantages over programming in binary (hexadecimal) notation:

Mnemonic operation codes are used to designate an operation.

Addresses of data and instructions may be assigned symbolic names that are used in subsequent instructions.

The programmer does not have to be concerned where the program will ultimately reside in memory.

The programmer may specify constant data in alphabetic or decimal format rather than binary format.

Symbolic programs are easily modified because additional statements may be inserted into an existing statement sequence.

2.2 THE ASSEMBLER PROGRAM

The statements written in symbolic assembler language must be translated into machine language before the processor can execute the instructions.

The conversion of the program from its symbolic representation to binary representation is performed by the assembler program. The assembler program translates the symbolic mnemonics to a machine language program. This conversion is called the assembly process.

The assembly process starts with the symbolic source program written by the programmer. An example of a source program is shown in figure 2-1. The statements may be punched on cards or paper tape for input to the assembler program or the statements may be input via a computer terminal keyboard. The source program deck or file is the primary input to the assembler program.

The source program contains two basic types of statements: machine instructions and assembly instructions. The assembler program processes each type differently.

Machine instructions are used to request the processor to perform a sequence of operations during program execution time. Machine instruction statements are a one-for-one symbolic representation of actual machine language instructions. The assembler program generates an equivalent machine instruction in the object program for each machine instruction statement.

Operands of machine instructions usually represent storage locations, registers, immediate data, or constant values. A machine instruction statement may be identified by assigning a name (label) to it. The value of the label is the address of the assembled machine instruction.

Assembler instructions are used to request the assembler program to perform certain operations during the assembly process. These operations assist the programmer in data and symbol definition, in checking and documenting the program, in controlling the assignment of storage addresses, in program sectioning and linking, in defining data and storage fields, and in controlling the assembler auxiliary functions to be performed by the assembler program. With few exceptions, assembler instructions do not result in the generation of any machine language code in the object program.


```

1.      REVISION-F 10/02/73      10/22  5:28PM%
2.      SSORT    SIMPLE SORT (08/01/73)  PAGE NUMBER  1
3.
4.
5.      1 0000      .TITLE SSORT,'SIMPLE SORT (08/01/73)'      4.
6.      2 0000      .GLOBL SSORT      ;CAN REFERENCE ENTRY POINT FROM      5.
7.      3 0000      ;SEPARATE ASSEMBLY      6.
8.      4 0000      ; SSORT SORTS A VECTOR OF SINGLE-WORD CONSTANTS INTO ASCENDING      7.
9.      5 0000      ; ORDER. CALLING SEQUENCE IS:      8.
10.     6 0000      ;      9.
11.     7 0000      ; JSR    SSORT      ;CALL      10.
12.     8 0000      ; .WORD  VECTOR      ;ADDRESS OF VECTOR      11.
13.     9 0000      ; .WORD  VECTOR+LENGTH-1      ;ADDRESS OF LAST WORD OF VECTOR      12.
14.    10 0000      ; ...      ;NORMAL RETURN      13.
15.    11 0000      ;      14.
16.    12 0000 0000 A FLAG: .WORD  0      ;IF NON-ZERO, SWAP MADE DURING PASS      15.
17.    13 0001 0002 T TAB:  .=.+1      ;VECTOR ADDRESS      16.
18.    14 0002 0003 T TABEND: .=.+1      ;SORT LIMIT      17.
19.    15 0003 0007 T REGS:  .=.+4      ;REGISTER SAVE AREA      18.
20.    16 0007 A1FC A SSORT: ST    0,REGS      ;SAVE REGISTERS      19.
21.    17 0008 A5FB A      ST    1,REGS+1      20.
22.    18 0009 A9FB A      ST    2,REGS+2      21.
23.    19 000A ADFB A      ST    3,REGS+3      22.
24.    20 000B 4600 A      PULL   2      ;OBTAIN ADDRESS OF PARAMETER LIST      23.
25.    21 000C 4200 A      PUSH   2      24.
26.    22 000D 4C00 A      LI     0,0      25.
27.    23 000E A1F1 A      ST     0,FLAG      26.
28.    24 000F 2E01 A      LD     3,1(2)      ;END OF VECTOR      27.
29.    25 0010 4BFF A      AISZ   3,-1      28.
30.    26 0011 ADF0 A      ST     3,TABEND      29.
31.    27 0012 8E00 A      LD     3,(2)      ;VECTOR ADDRESS      30.
32.    28 0013 ADED A      ST     3,TAB      31.
33.    29 0014 8300 A LOOP:  LD     0,(3)      ;GET A VALUE      32.
34.    30 0015 E301 A      SKG     0,1(3)      ;COMPARE AGAINST NEXT VALUE      33.
35.    31 0016 2105 A      JMP     TEST      ;VALUES IN ORDER      34.
36.    32 0017 8701 A      LD     1,1(3)      ;SWAP VALUES      35.
37.    33 0018 A301 A      ST     0,1(3)      36.
38.    34 0019 A700 A      ST     1,0(3)      37.
39.    35 001A 4D01 A      LI     1,1      ;SET SORT FLG NON-ZERO      38.
40.    36 001B A5E4 A      ST     1,FLAG      39.
41.    37 001C 4B01 A TEST: AISZ   3,1      ;INCREMENT TABLE POINTER      40.
42.    38 001D EDE4 A      SKG     3,TABEND      ;FINISHED THIS PASS?      41.
43.    39 001E 21F5 A      JMP     LOOP      ;NO      42.
44.    40 001F 81E0 A      LD     0,FLAG      ;YES - DID WE MAKE A SWAP?      43.
45.    41 0020 4800 A      AISZ   0,0      44.
46.    42 0021 2101 A      JMP     .+2      ;YES - CONTINUE SORT      45.
47.    43 0022 2104 A      JMP     OUT      ;NO - SORT DONE      46.
48.    44 0023 4C00 A      LI     0,0      ;INITIALIZE FOR NEXT PASS      47.
49.    45 0024 A1DB A      ST     0,FLAG      48.
50.    46 0025 8DD8 A      LD     3,TAB      49.
51.    47 0026 21ED A      JMP     LOOP      50.
52.    48 0027 81DB A OUT:  LD     0,REGS      ;RESTORE REGISTERS      51.
53.    49 0028 85DB A      LD     1,REGS+1      52.
54.    50 0029 89DB A      LD     2,REGS+2      53.
55.    51 002A 8DDB A      LD     3,REGS+3      54.
56.      REVISION-F 10/02/73      10/22  5:28PM%
57.      SSORT    SIMPLE SORT (08/01/73)  PAGE NUMBER  2
58.
59.
60.      52 002B 0202 A      RTS     2      55.
61.      53 002C      .END      56.
62.
63.
64.      ***** 0 ERRORS IN ASSEMBLY *****
65.      REVISION-F 10/02/73      10/22  5:28PM%
66.      SSORT    SIMPLE SORT (08/01/73)  PAGE NUMBER  3
67.
68.
69.      FLAG  LOOP  OUT  REGS  SSORT  TAB  TABEND  TEST
70.      0000 T 0014 T 0027 T 0003 T 0007 T 0001 T 0002 T 001C T
71.
72.
73.      E4C6      %%0
74.

```

Figure 2-2. Example of IMP-16 Cross Assembler Listing

Operands of assembler instructions provide the information needed by the assembler program to perform the designated operation.

Two outputs are generated as a result of running a source program (programmer generated statements) through the assembler program: (1) an object program consisting of actual machine instructions corresponding to the source program statements, and (2) a program listing showing source statements side by side with the object code instructions created from the statements. Most programmers work with the program listing once it is available. A sample cross assembler listing is shown in figure 2-2.

As a source program is assembled, it is analyzed for errors in the use of the assembler language. Any detected errors are indicated on the program listing to assist the programmer in debugging.

The flowchart in figure 2-3 shows the relationship of the assembler program to the programming process.

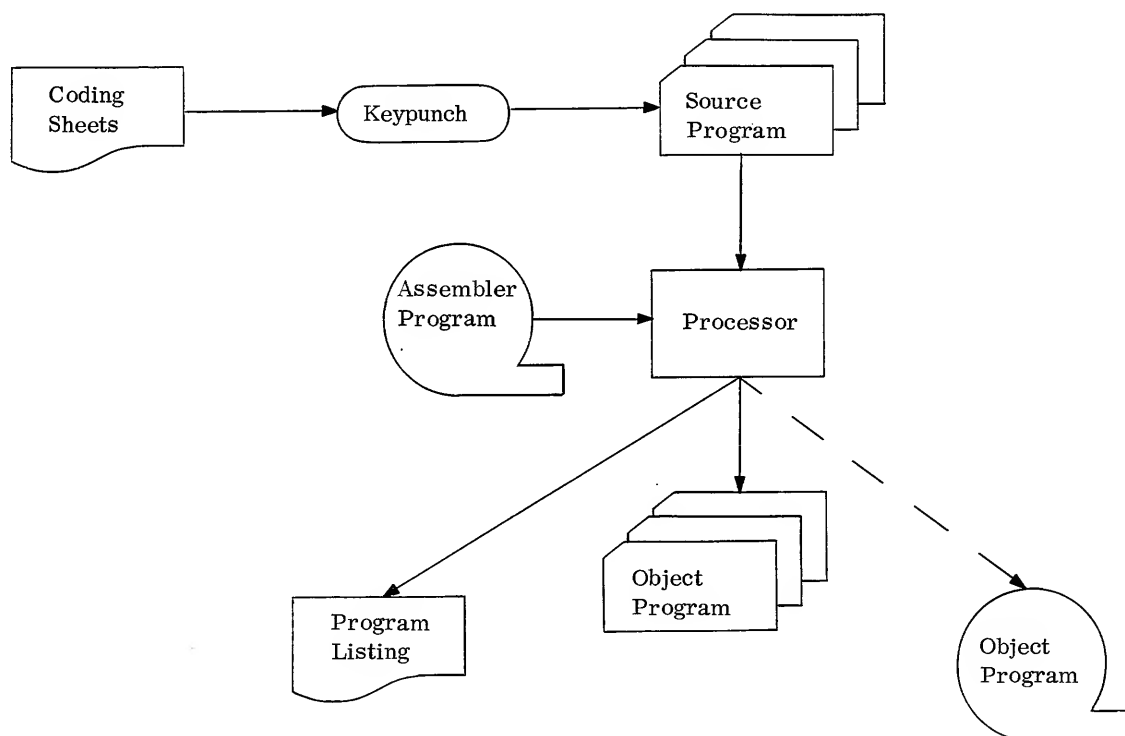


Figure 2-3. Programming Process

Some assembler programs, called one-pass assemblers, completely process the symbolic code during one pass. Others make two passes through the source code. On the first pass, the assembler program determines how many words of storage are required for each statement and assigns a beginning value for the first location in every statement line. It generates the machine language program and assembly listing during pass two. The IMP-16 Cross Assembler Program is a two-pass assembler. The IMP-16 Resident Assembler requires three passes, producing a listing on pass-2 and a binary tape on pass-3.

In summary, a program goes through six basic processes: (1) design of flowchart, (2) coding of source statements, (3) assembly run, (4) debugging, (5) test run, and (6) production run. A Programmers Checklist is provided in appendix D to assist the programmer in desk checking for problems before the assembly run.

2.3 ASSEMBLER PROGRAMMING AIDS

2.3.1 Data Representation

Decimal, hexadecimal, or character representation of machine-language binary values may be used in source statements. Since the byte is the smallest unit of memory storage that can be addressed, the 8-bit code permits arrangement of the bits into 256 different configurations. These 256 unique characters can then represent all letters of the alphabet, the numbers 0 to 9, punctuation marks, and additional special symbols and control characters.

2.3.2 Binary Fixed-Point Format

Fixed-point instructions perform binary arithmetic on binary fixed-point formatted operands. Fixed-point numbers consist of a 1-bit sign followed by a 15-bit integer field.

It is called fixed point because the processor interprets the number as a binary integer; that is, the point is to the right of the least significant position. The programmer has the responsibility for keeping track of an assumed point within a field.

All fixed-point operands are treated as signed integers. True binary notation with a sign bit of zero is the representation of positive numbers. Twos-complement notation with a sign bit of one is the representation of negative numbers. To obtain the twos complement of a number, the value of each bit is complemented and a one is added to the low-order bit.

2.3.3 Memory Addressing

During execution of a program, instructions and data are stored in successive memory locations in the main memory of the processor. It is characteristic of stored-program processors that they may treat the contents of a particular memory location as an instruction at one time and as data at another time.

It has been noted previously that one of the advantages of an assembler language is the use of symbolic addresses rather than actual addresses.

An "actual address" refers to the number identifying the location in memory that is actually passed by the processor to memory in order to read or write a particular location at execution time.

A "symbolic address" is assigned by the programmer when writing the source statement and is used to identify a particular assembler instruction or data word.

During the assembly process the assembler program maintains a "location counter" to tell where it is in the assembly process and to use in assigning numeric values to symbolic addresses. In order to understand how the assembler keeps track of addresses assigned to symbols, it is necessary to understand the concept of the location counter.

Whenever a symbol appears in the "label field" of a statement, it is assigned a value that may be considered to be an address in the object program. This value is called the location counter value. As each machine instruction or data area is assembled, the location counter is incremented by the length of the assembled item. Thus, it always points to the location of the next available storage area in memory.

Since the location counter is initialized to a value of zero at the beginning of each section in a program during the assembly process, the location counter value assigned to the symbolic address is the relative location word of the field being assembled to the first field that was assembled in the section; that is, its "relative address." Then, when the program is loaded into main memory for execution, the relative addresses are mapped onto actual addresses (main memory addresses).

The assembler system compiles a table containing all of the symbols that appear in the label field. A specific memory address for each label is stored in the symbol table. References to symbols cause the Assembler Program to interrogate the symbol table for the address of the field referred to in the statement.

Note that on the first pass, the assembler program determines how many words are required for each statement and constructs a symbol table containing all symbols used. On the second pass, the operator and operand fields for all statement lines are constructed. Using the values assigned in the symbol table during the first pass, the location counter value assigned to the symbol can be substituted whenever the symbol appears as an operand. Likewise, all numeric values are replaced with binary values and the machine format is constructed.

2.3.4 Relocatability

The object programs produced by the assembler program may be produced in absolute and/or relocatable formats. Absolute data can only be loaded in memory, for execution, at the exact location for which it was assembled. Relocatable data may be loaded for execution in any suitable area in memory. The loading of relocatable data has no relation to where the data were stored when assembled.

2.3.5 Linking

The programmer may define symbols in one program and refer to these symbols in another, thus providing symbolic linkages between independent programs. This permits reference to data and/or transfer of control between programs.

Chapter 3

OVERVIEW OF CPU

The IMP-16 is a 16-bit parallel processor containing read/write memory, read-only memory, register and arithmetic logic units, and input/output control. Up to 65,536 16-bit words of semiconductor memory are supported.

This section briefly reviews those parts of the IMP-16 with which the programmer is primarily concerned. Detailed information on the IMP-16 configuration is covered in the appropriate IMP-16 Users Manual.

3.1 OPERATIONAL FEATURES

Word Length	16 bits
Instruction Set	40 in basic set; 21 in extended set (implemented by CPU-resident microprogram)
Arithmetic	Parallel, binary, fixed point, twos complement
Addressing	Direct and indirect addressing Absolute Relative to Program Counter Indexed
Typical	Register-to-Register addition -- 4.9 microseconds
Instruction-	Memory-to-Register addition -- 8.4 microseconds
Execution	Register input/output -- 10.5 microseconds
Speeds (IMP-16L)	

3.2 ARITHMETIC AND LOGIC UNITS REFERENCED BY IMP-16 ASSEMBLER LANGUAGE AND PROGRAM

The units referenced in the discussion of the IMP-16 assembler language and program are discussed below and are shown in a block diagram in figure 3-1.

3.2.1 Last-In/First-Out Stack (LIFOS)

The IMP-16 has a hardware stack that data may be stored in or retrieved from on a last-in/first-out basis. The stack is 16 words deep and is accessible through the top location. As a data word is entered into the stack, the contents of the top location and each other location are pushed downward to the next lower level; if the stack is full, the word in the bottom location is lost. Conversely, the contents of the top location are pulled from the stack during retrieval of a data word; the top location and each lower location are replaced by the contents of the next lower location, and zeros are entered into the bottom location.

The stack is used primarily for saving status during interrupts and for saving subroutine return addresses. It may also be used for temporary storage of data using the PUSH, PULL, and XCHRS instructions.

3.2.2 Status Register

There are 16 RALU status flags. These flags may be pushed onto the stack (saved) or may be loaded from the stack (restored). During such operations, the flags are configured as a 16-bit word; the L (Link), CY (Carry), and OV (Overflow) flags are the first, second, and third most significant bits, respectively, and the remaining 13 flags are assigned various functions in the various members of the IMP-16 family. The specific uses of certain flags are discussed in chapter 6.

3.2.3 Program Counter (PC)

The program counter (PC) holds the address of the next instruction to be executed. It is incremented by 1 immediately following the fetching of each instruction during execution of the current instruction. When there is a branch to another address in the main memory, the branch address is set into the program counter. A skip instruction merely increments the program counter by 1, thus causing the one instruction to be skipped.

3.2.4 Memory Data Register (MDR) and Memory Address Register (MAR)

These two registers hold data and addresses, respectively, for instructions being executed.

3.2.5 Working Registers

The 16-bit accumulators, 0, 1, 2, and 3 are used as working registers for data manipulation. Data words may be fetched from memory to an accumulator or stored from an accumulator into memory. The particular accumulator to take part in an operation is specified by the programmer in the appropriate instruction.

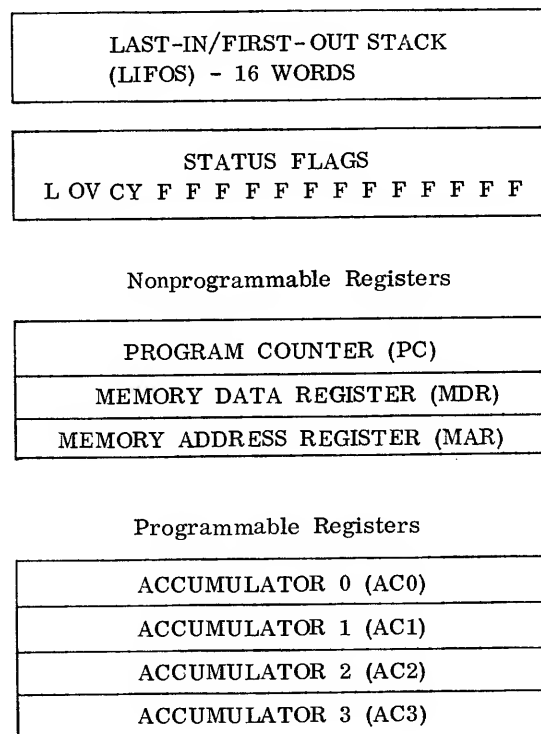


Figure 3-1. Arithmetic and Logic Units Referenced by IMP-16 Assembler

Chapter 4

IMP-16 ASSEMBLER LANGUAGE

4.1 IMP-16 ASSEMBLER CODING CONVENTIONS

Source statements may contain from one to five entries: Label, Operation Mnemonic, Operand(s), Comment, and Identification. The fields must be entered in the above order with one or more blanks separating each field. The IMP-16 assembler program accepts free-form statements allowing the programmer to disregard the boundaries specified on the coding form, if he chooses. A sample coding form is shown in figure 4-1.

4.1.1 Label Field

The label field is an optional field and may contain a symbol used to identify the statement in other statements. Each label must be terminated by a colon (:). More than one label may appear in the label field.

4.1.2 Operation Field

The operation field is a mandatory field that contains a mnemonic operation code defining the assembler or machine operation.

4.1.3 Operand Field

Entries in the operand field identify and describe the data to be acted upon by the statement. One or more operand entries may be needed, depending on the operation code.

1. If more than one operand is present, the operand entries must be separated by commas.
2. Operands may not contain embedded blanks unless the entry specifies a data string containing blanks.

4.1.4 Comment Field

Comments are optional descriptive notes that are printed on the program listing for programmer reference. Comments should be included throughout the program to explain subroutine linkages, assumptions made, formats of inputs processed, and so forth. A comment may follow a statement or it may be entered on a separate statement line(s), since it has no effect in the assembled program and is only printed on the listing.

The following conventions apply to comments:

1. A comment must be preceded by a semicolon (;).
2. All valid characters, including blanks, may be used in comments.
3. Comments cannot extend beyond column 72, but further comment may be entered on the following line (preceded by a semicolon).

[illegible][illegible]

Figure 4-1. Sample Coding Form

4.2.1 Label Entry

Labels are symbolic terms (see below) and must conform to their structure. The following rules apply to labels:

1. A label may contain from one to 32 alphanumeric characters and must conclude with a colon (:), for example, TABEND:.
2. Blanks cannot appear within the label.
3. For nonlocal labels, the first six characters must be unique. For local labels, the first five characters must be unique.

4.2.2 Operation Mnemonic Entry

Valid operation mnemonics are defined in detail in chapters 7, 8, and 9. Operation mnemonics are classified as Assignment, Directive, and Instruction codes. Assignment Statements assign a value to a symbol. Directive Statements control the process of program assembling and may generate data. Instruction Statements define the machine operations necessary to execute the program.

4.2.3 Operand Entry

An Operand Entry is composed of one or more terms.

All terms represent a value. The value may be assigned by the assembler program during assembly (symbols) or it may be inherent in the term itself (self-defining). An arithmetic combination of terms is reduced to a single value by the assembler program.

The various types of terms are defined below.

Symbolic Terms

Symbols are the most common means of referencing address locations or arbitrary values. Symbols are defined (assigned values) by one of three methods:

1. By appearing in the "label field" in a statement.
2. By using an "assignment statement" to assign a specific value to a symbol.
3. By using a ".FORM directive statement" to assign a value to a symbol.

A symbol that is used to reference a location or value (but not a .FORM symbol) may be further identified as a global symbol permitting other programs to access its value.

The value assigned to the symbol is the address of the instruction, data, or storage location named by the symbol. If the address of an item changes upon program relocation, the symbol is considered a "relocatable term." If the address does not change upon program re-location, the symbol is an "absolute term."

Symbol construction must meet the following restrictions:

1. A symbol may contain one or more alphanumeric characters, the first of which must be either a letter or a dollar sign (\$).
2. Although up to 32 letters may be included, only the first six letters are recognized by the assembler program. Therefore, the programmer must ensure that a long symbol is unique in the first six characters.
3. If the first character in the symbol is a dollar sign (\$), the symbol is defined as a local symbol. The .LOCAL operator allows the programmer to specify that local symbols appearing between two .LOCAL directive statements have a certain meaning only within that region of the program. This enables the programmer to use common mnemonics throughout a program without causing a conflict of names. Note: A long local symbol must be unique in its first five characters.
4. No special characters or embedded blanks may appear within a symbol.
5. Symbol values cannot exceed a positive value of 65,535 or a negative value of 32,768.

Several examples of symbols follow:

<u>Legal Symbols</u>	<u>Illegal Symbols</u>
\$ABC	LONGSYMBOL1
LONGSYMBOL	LONGSYMBOL2
\$AB2	2AB
\$2	#CDE
XYZ	XYZ\$
\$ABCDEF	\$ABCDE
\$ABC2EF	\$ABCDF

A program assembled on the IMP-16 cross assembler may contain 700 symbols. One assembled on the resident assembler, in 4K of memory, may contain approximately 25 symbols. A program assembled on 8K of memory may contain approximately 825 symbols.

Self-Defining Terms

A self-defining term is one whose value is inherent in the term. The assembler program does not assign a value to the term but uses the term itself as the value to be assembled.

Self-defining terms are used to specify immediate data, addresses, registers, and I/O information to the assembler program. Three types of self-defining terms are available: decimal, hexadecimal, and character.

A decimal self-defining term is specified by a nonzero first digit followed by the number of digits necessary to make the number. The range of decimal numbers is -32768 through 65535. Examples include: 32671, 10, 5, -600.

A hexadecimal self-defining term is specified in one of two ways. The term may be preceded by X' or the term can start with a leading zero. The range of hexadecimal numbers is -8000₁₆ to FFFF₁₆. Examples include: X'A2, 0A2, X'1234, 01234, 0, X'0, 0ABCD, X'ABCD, -X'7F.

A character self-defining term is specified as a string. A string is a series of characters or a single character enclosed in single quote marks (for example, 'THIS IS A STRING'). All letters and special characters may be specified in a string. If a single quote mark is part of the character string, it should be immediately preceded by another single quote mark, for example, 'DON'T DO IT' represents DON'T DO IT. String characters are translated to ASCII code (see appendix A) in memory with each character occupying 8 bits. If a string contains an odd number of characters, it will be padded with a trailing blank. An empty string (for example, '') will cause generation of a word containing two blanks.

EXPRESSIONS

Operand entries consisting of either a single term or an arithmetic combination of terms are called expressions. Expressions are either simple or multiterm. Simple expressions are single terms such as a symbol or a self-defining term. Multiterm expressions are simple expressions that are combined by arithmetic operators for evaluation by the assembler program. Multiterm expressions are formed in the same manner as normal arithmetic expressions and are evaluated by the assembler program in a strict left-to-right order without regard to treating a particular operator before any other. Parentheses are not permitted.

The following table lists the arithmetic operators available for forming multiterm expressions.

<u>Arithmetic Operator</u>	<u>Function</u>	<u>Type</u>
+	Addition	Binary
-	Subtraction	Unary or binary
*	Multiplication	Binary
/	Division	Binary
%	Unary	Logical NOT
&	Binary	Logical AND
[Binary	Logical OR

A unary operator operates upon one operand and appears in the format "op opnd." For example, -9. A binary operator operates upon two operands and appears in the format "opnd₁ op opnd₂." For example, A&B.

In addition to representing a value, all expressions have a mode. Mode reflects the relocation of the assembled expression in memory at load time. The programmer creates a source program in sections producing a load module that is absolute, base sector relocatable, or top sector relocatable or a combination of the three (see 5.2); the assembler program allocates the assembled program to main memory in sectors. Therefore, mode may be absolute, base sector relocatable, or top sector relocatable.

An expression is absolute if its value is unaffected by program relocation. An absolute expression may be a single term or the result of an arithmetic combination of terms.

An expression with a top or base sector relocatable mode is one whose value would change by n if the section which it references is relocated n words away from its originally assigned storage area.

Self-defining terms always have an absolute mode. The mode of symbolic terms depends on where the term was coded in the source program. Whenever a new section is encountered during the assembly process, the location counter is set to the mode of that section. For example:

1. A symbol used as a label has the mode of the location counter at the time that label was encountered by the assembler program.
2. A symbol given a value by an assignment statement has the mode of the expression on the right side of the assignment statement.
3. A global symbol has an external mode, and may not appear in a multiterm expression.

The mode of the resulting value from a multiterm expression is determined by the modes of the terms within the expression and the arithmetic operators used to combine the terms. (When a character string is included in an expression, it is treated as though it is two characters long. If more than two characters are in the string, the left-most pair is retained and the excess is ignored. If there is only one character in the string, it is treated as one character followed by a blank.)

Table 4-1 lists legal mode and arithmetic operator/term combination. This table is used to determine the mode assigned by the assembler program to the result of a multiterm expression.

Table 4-1. Expression Classification Table

Multiterm Mode		New Expression Mode			
Left Term	Right Term	Operator L + R	Operator L - R	Operator L x R	Comments
A	A	A	A	A	A = Absolute B = Base Sector T = Top Sector x = %, *, /, & or ! I = Illegal
A	B	B	I	I	
A	T	T	I	I	
B	A	B	B	I	
T	A	T	T	I	
B	B	I	A	I	
T	T	I	A	I	
B	T	I	I	I	
T	B	I	I	I	

How to use table 4-1:

1. Read the expression from left to right, a term at a time. Determine the present mode of the first term. Initially the L Term refers to the term on the left side of the arithmetic operator. The resulting evaluation mode when the two terms are combined becomes the L Term for the next combination. The R Term refers to the term on the right side of the arithmetic operator.

2. Find the proper combination of modes (multiterm mode column).
3. Select the appropriate arithmetic combination of the L Term and the R Term and follow that column down to the row selected in Step 2 above. The new expression mode is indicated in that row.

Note: x stands for %, *, /, !, or %.

4. If a unary operator (- or %) appears at the beginning of an expression, assume an absolute zero appears to the left of the unary operator. In other words, the L Term is absolute and the R Term assumes the mode of the term to the right of the unary operator. Then consult the table in the normal way.
5. An external symbol (a global symbol that can be referenced by other programs) is only valid when used as a simple expression. This means that an external symbol cannot appear in a multiterm expression.

Given the following terms and related modes, some examples of the use of the Expression Classification Table follow:

<u>Terms</u>	<u>Mode</u>
A1, A2, A3	Absolute
B1, B2	Base Sector Relocatable
T1, T2, T3	Top Sector Relocatable

1. A1 + B2

A1 (L Term) is absolute mode and B2 (R Term) is base mode (line 2). Following to the right, the L + R column indicates a new expression mode of base (B).

2. T3 + 6 - T1

Expressions are evaluated in a strict left-to-right order so T3 + 6 is evaluated first. T3 (L Term) is top mode and the constant 6 is absolute mode. The combination is found in line 5. L + R determines that the mode of T3 + 6 is also top.

Since the resulting evaluation is top mode, the L Term for the next evaluation is top.

$$\underbrace{T3 + 6}_{\text{top}} - T1_{\text{top}}$$

The mode of T1 (R Term) is top). Entering L - R (line 7) gives absolute as the mode for the expression as a whole.

3. T2 - A2 + T3 - A3 + B1

First evaluate T2 - A2. T2 (L Term) is top mode while A2 (R Term) is absolute. The combination (top, absolute) is found in line 5 under operator L - R. Top is given as the resulting mode (new L Term).

$$\underbrace{T2 - A2}_{\text{top}} + T3_{\text{top}} - A3 + B1$$

T3, the new R Term is top. Therefore, line 7 is selected. Under L + R, the expression is illegal and will not be processed further.

4.3 DATA REPRESENTATION

Data are represented in the IMP-16 in twos-complement integer notation. In this system, the negative of a number is formed by complementing each bit in the data word and adding 1 to the complemented number. The sign is indicated by the most significant bit. In the 16-bit word of the IMP-16, when bit 15 is a "0," it denotes a positive number; when it is a "1," it denotes a negative number. Maximum number ranges for this system are $7FFF_{16}$ ($+32767_{10}$) and 8000_{16} (-32768_{10}) for single-precision operations. For double-precision operations, the first consecutive word addressed in memory contains the high-order part, and the next consecutive word contains the low-order part of a double-precision number. Bit 15 of the high-order part is the sign bit.

Chapter 5

ADDRESSING

5.1 REGISTERS

The four working registers (accumulators) are addressed by placing the numeric address (0, 1, 2, or 3) in the operand field. The addresses and corresponding accumulators are as follows:

0	Accumulator 0	(AC0)
1	Accumulator 1	(AC1)
2	Accumulator 2	(AC2)
3	Accumulator 3	(AC3)

NOTE

The "assignment statement" (chapter 8) may be used to assign symbolic addresses to the working registers.

Programs are commonly written using the working registers as follows:

AC0 — primary data-handling register
 AC1 — secondary data-handling register
 AC2 — base register for indexed addressing
 AC3 — base register for indexed addressing

5.2 MEMORY

The addressable memory of the IMP-16 is 0 to 65535. Main memory is divided into two areas. Because the majority of the operations in the instruction set have an address field of only 8 bits, and therefore, allow explicit addressing of words 0 to 255 in memory, the "base sector" comprises locations 0 to 255. The "top sector" comprises the remainder of memory.

Based on the program section directive statements coded by the user, the IMP-16 assembler program allocates a program to main memory in three sectors or parts: base sector relocatable, top sector relocatable, and absolute. The portion of the program allocated to the base sector is directly addressable from any location in main memory. The portion of the program allocated to the top sector is addressed directly or indirectly. Data assigned an absolute address is placed in either the base sector or top sector. No check is made to determine whether the data in absolute locations will interfere with the base sector or top sector.

5.3 LOCATION COUNTER

The "location counter" assigns relative addresses to program statements. It is similar to the CPU program counter, which contains the main memory address of the next instruction to be executed at execution time. As each machine instruction or data area is assembled, the location counter is incremented by the length of the assembled item. Thus, it always points to the location of the next available storage location in memory. If the statement is named by a label, the value assigned to the label is the value of the location counter.

The IMP-16 assembler program maintains separate location counter addresses for each sector (absolute, base, and top). The assembler program normally begins assembly of each sector with the corresponding location counter mode initialized to a value of zero. Source statements in each sector are assigned addresses from the location counter set for the appropriate sector. If a sector is interspersed throughout a program, whenever the sector is re-encountered, the location counter is reset to reflect the appropriate sector mode and assumes the value that was its last value when assembling in that sector.

The location counter mode is controlled by the "program section directives" (chapter 9), and the "assignment statement" (chapter 8) is used to change the value.

5.4 METHODS OF ADDRESSING

Three methods of accessing data by an instruction are available: immediate, direct addressing, and indirect addressing.

5.4.1 Immediate Addressing

A statement that contains the value of its operand in the operand field itself has an immediate address. This method is limited to certain operation mnemonics. All immediate operands are absolute since their value does not change when the program is relocated.

5.4.2 Direct Addressing

A direct-address operand specifies the address of the location in memory whose contents are used during execution of an instruction. Direct addresses fall into three categories: base page, program counter relative, and indexed.

Base Page

Base-page addresses refer to the base sector and consequently can only address bytes 0 to 255 in memory. Base page addresses are directly accessible from any location in memory. The operand specifying the address must have a value in the 0 through 255 range.

NOTE

The assembler program automatically assigns value 0 to the index register field in the machine instruction format, thus specifying base page address.

Program Counter Relative Addressing

A program-counter-relative address is formed by adding the current contents of the program counter (PC) to the value specified in the operand field. The value is treated as a signed number since its sign bit (bit 7) is propagated to bits 8 through 15. This permits program-counter-relative addressing -128 and +127 locations from the PC value. Note, however, at the time the program-counter-relative address is calculated, the program counter has already been incremented and is pointing to the next memory location. Therefore, the actual addressing range is -127 to +128 from the current instruction.

NOTE

The assembler program automatically assigns value 01 to the index register field in the machine instruction format, specifying program-counter-relative addressing.

The programmer cannot explicitly specify program counter relative addressing. If the programmer specifies a base-sector reference without indexed addressing in the operand field, the assembler program automatically indicates the base page address mode. If a top-sector reference is indicated, the assembler program attempts program-counter-relative addressing.

If both modes fail, an addressing error occurs.

Indexed Addressing

Indexed addressing enables the programmer to address any location in the 65K memory by utilizing a base register addressing scheme. Base-register addressing requires the designation of an accumulator (containing a base address) and a displacement value for specifying a storage location. The assembler adds the contents of the register to the number formed from the displacement value to yield the address. For example, assume that AC2 contains a base address of 300, and the displacement value is 120. The displacement is added to the base address and the result is an address of 420.

Only accumulators 2 and 3 may be used as base (index) registers, and a base value must be previously assigned to the register before it is used in an address.

IMP-16 indexed addressing allows the programmer to address 256 words around the base address; that is, the base address represents the middle of a floating page. The displacement can be -128 through +127 from the base.

An indexed address operand contains the displacement immediately followed by the index register address in parentheses. The register address must be absolute and evaluate to 2 or 3 (accumulator 2 or 3). The displacement number is treated as a signed 8-bit number from -128 to +127.

5.4.3 Indirect Addressing

An indirect address operand specifies the address of a memory location that holds the address of the data to be used as the effective address by the instruction. Indirect addresses fall into three categories: base page, program counter relative, and indexed. The address is calculated using the same methods used for direct addresses with one exception (See JINT, chapter 7). Indirect addressing is limited to certain operations and is specified by a @ before the displacement value in the operand field.

5.4.4 Double-Word Addressing

Ten of the 60 instructions use a double-word machine instruction format. These instructions utilize direct and indirect addressing methods exactly as the single-word instruction. The 16-bit displacement field makes all of memory directly addressable, although indexed addressing may be used if desired. It is important to note that with program-counter-relative addressing, the program counter contains the address of the displacement word (second byte) of the instruction. The instructions are Multiply (MPY), Divide (DIV), Double Precision Add (DADD), Double Precision Subtract (DSUB), Load Byte (LDB), Store Byte (STB), Load Left Byte (LLB), Load Right Byte (LRB), Store Left Byte (STB), and Store Right Byte (SRB).

NOTE

IT IS RECOMMENDED THAT PROGRAM-COUNTER-RELATIVE ADDRESSING NOT BE USED WITH BYTE INSTRUCTIONS.

The address operand gives the data required to calculate an effective memory address (EA) at execution time. Operand fields specifying addresses are summarized in the following table.

Table 5-1. Address Operands

Type	Operand Field	Address Calculation
Direct Base Page Direct PC-Relative	displacement	EA = disp EA = disp + PC
Indirect Base Page Indirect PC-Relative	@displacement	EA = disp EA = disp + PC
Direct Indexed	displacement (index)	EA = disp + xr
Indirect Indexed	@displacement (index)	EA = disp + xr

EA - effective address specified by the instruction. The contents of the effective address are used during execution of an instruction.

disp - stands for displacement value and is an 8-bit, signed twos-complement number except when base page address is specified.

PC - program counter.

xr - index register (AC2 or AC3).

Operator Address Classes

Instruction statement memory-reference operators are separated into seven classes according to addressing capability. The classes are defined below.

Class 1 ADD, SUB, SKG, SKNE, AND, OR, SKAZ, ISZ, DSZ

- a. May directly address all of base sector (0-255).
- b. May use indexed addressing with displacement range -128 through 127.
- c. If not indexed, assembler program will attempt program-counter-relative addressing for top sector reference.

Class 2 LD, ST, JMP, JSR

- a. May directly address all of base sector (0-255).
- b. May use indexed addressing with displacement range -128 through 127.
- c. If not indexed, assembler program will attempt program-counter-relative addressing for top-sector reference.
- d. May use indirect addressing to address all of memory.

- e. For a top-sector reference, if the instruction is not indexed or marked indirect already, and if it is not possible to use program-counter-relative addressing, the assembler will force indirect addressing through a base-page pointer. The cross assembler will generate up to 125 pointers in base sector; the resident assembler up to 50 pointers. Generation of a base-page pointer may be avoided by marking the instruction indirect via an explicit top-sector pointer.

Class 3 MPY, DIV, DADD, DSUB (Extended Instruction Set)

- a. Instruction length is two words, so instruction may directly address all of memory.
- b. May use indexed addressing to address all of memory.

Class 4 LDB, LLB, LRB, SLB, SRB, STB (Extended Instruction Set)

- a. May address the memory range 0 through $7FFF_{16}$.
- b. May use indexed addressing.
- c. Program-counter-relative addressing is not available for these instructions.

Class 5 JINT (Extended Instruction Set)

- a. May address all of memory indirectly via a 16-word jump table in locations 120_{16} through $12F_{16}$.
- b. May not use indexed addressing.
- c. Program-counter-relative addressing is not available for this instruction.

Class 6 JSRI

- a. May address the memory range $FF80_{16}$ through $FFFF_{16}$.
- b. May not use indexed addressing.
- c. Program-counter-relative addressing is not available for this instruction.

Class 7 JMPP, JSRP (Extended Instruction Set)

- a. May address all of memory directly.
- b. May not use indexed addressing.
- c. Program-counter-relative addressing is not available for these instructions.

Assembler Program Generation of Instructions Containing Addresses

Tables 5-2 through 5-7 illustrate the various addressing formats and define the type of addressing structure developed by the assembler program for 1-word memory reference instructions.

In order to use the tables, the address type must be determined and the proper table selected. The following address types apply:

Direct Base Page or Program-Counter Relative ----- table 5-2
 Indirect Base Page or Program Counter Relative ---- table 5-3
 Direct Indexed ----- table 5-4
 Indirect Indexed ----- table 5-5
 Direct Indexed Without Displacement Value----- table 5-6
 Indirect Indexed Without Displacement Value ----- table 5-6

The table is entered in the appropriate expression mode, and the first applicable case listed under "assembler interpretation" is determined. The following results are given:

1. The action taken by the assembler program
2. The mode of the generated object code

Table 5-2. ASSEMBLER EXECUTION OF DIRECT BASE PAGE OR PROGRAM-COUNTER-RELATIVE ADDRESS

Example: L R1, TABLE		
Address Expression Mode	Assembler Program Interpretation	Object Code Mode for Instruction
absolute	(1) $0 \leq \text{expression} \leq 255$ A reference to the base sector is assumed and an absolute address is generated.	absolute
	(2) If the expression is the same mode as the location counter and $-128 \leq \text{location counter} + 1 - \text{expression} \leq 127$, then, program counter relative addressing is used.	absolute
	(3) $\text{expression} > 255$ <u>and</u> the instruction will permit indirect addressing. A reference to the top sector is assumed. A pointer to the address is established in the base sector (in a section immediately after the top of the base sector section for the current assembly). A reference to the pointer is made, and the instruction is forced indirect.	base sector relocatable (pointer object code mode is absolute)
	(4) $\text{expression} > 255$ and the instruction will <u>not</u> permit indirect addressing. ADDRESSING ERROR	
base sector relocatable	(1) $0 \leq \text{expression} \leq 255$ A relocatable base page reference is assumed.	base sector relocatable
	(2) $\text{expression} > 255$ ADDRESSING ERROR	

Table 5-2. ASSEMBLER EXECUTION OF DIRECT BASE PAGE OR PROGRAM-COUNTER-RELATIVE ADDRESS (Cont.)

Address Expression Mode	Assembler Program Interpretation	Object Code Mode for Instruction
top sector relocatable	<p>(1) Location Counter Top Sector Mode</p> <p>(a) If $-128 \leq \text{location counter} + 1 - \text{expression} \leq 127$ then an index through the location counter instruction addressing mode is generated.</p> <p>(b) If the expression is outside the above limits and indirect addressing is allowed, a pointer is established in the base sector. The instruction is forced to indirect addressing mode, and a reference is made to the base sector where the pointer resides.</p> <p>(c) If neither of the above conditions exist, there is an ADDRESSING ERROR</p>	<p>absolute</p> <p>base sector relocatable</p> <p>(pointer object code mode is top sector (relocatable))</p>
	<p>(2) Location Counter Base Sector or Absolute Mode.</p> <p>Rule (1)(b) above is tried. If the instruction cannot be made indirect, there is an ADDRESSING ERROR</p>	
external	The external must be in the base sector unless the instruction is LD, ST, JMP, or JSR; in which case it can be located anywhere in memory.	external

Table 5-3. ASSEMBLER EXECUTION OF INDIRECT BASE PAGE OR PROGRAM-COUNTER-RELATIVE ADDRESS

Example: L R1, @ TABLE		
Address Expression	Assembler Program Interpretation	Object Code Mode for Instruction
any type	If the expression is the same mode as the location counter and $(-128 \leq \text{location counter} + 1 - \text{expression} \leq 127)$, then, location-counter-relative addressing is used.	absolute
absolute	<p>(1) $0 \leq \text{expression} \leq 255$</p> <p>A reference to the base sector is assumed, and an absolute reference is generated.</p>	absolute
	<p>(2) $\text{expression} > 255$</p> <p>ADDRESSING ERROR</p>	
base sector relocatable	<p>(1) $0 \leq \text{expression} \leq 255$</p> <p>A relocatable base page reference is assumed.</p>	base sector relocatable
	<p>(2) $\text{expression} > 255$</p> <p>ADDRESSING ERROR</p>	

Table 5-3. ASSEMBLER EXECUTION OF INDIRECT BASE PAGE OR PROGRAM-COUNTER-RELATIVE ADDRESS (Cont.)

Address Expression Mode	Assembler Program Interpretation	Object Code Mode for Instruction
top sector relocatable	(1) Location counter top sector mode <u>and</u> $-128 \leq \text{location counter} + 1 - \text{expression} \leq 127$ A location-counter-relative instruction addressing mode is used.	absolute
	(2) All other cases ADDRESSING ERROR	
external	The external symbol must be in the base sector.	external

Table 5-4. ASSEMBLER EXECUTION OF DIRECT INDEXED ADDRESS

Example: LD R1, LABEL (R3)		
First Address Expression Mode	Assembler Program Interpretation	Object Code Mode for Instruction
absolute	(1) $-128 \leq \text{expression} \leq 127$ The value of the expression is placed in the displacement field. It will be added to the value on the specified index register at execution time.	absolute
	(2) All other cases ADDRESSING ERROR	
base sector relocatable	(1) $-128 \leq \text{expression} \leq 127$ The value of the expression is placed in the displacement field. It will be added to the value in the specified index register at execution time. Because the value is relocatable by the loader, care must be taken to ensure that the result is less than 127.	base sector relocatable
	(2) All other cases ADDRESSING ERROR	
top sector relocatable	ADDRESSING ERROR	
external	The external must be located in the base sector in locations 0 through 127.	external

Table 5-5. ASSEMBLER EXECUTION OF INDIRECT INDEXED ADDRESS

Example:	LD	R1, @LABEL (R3)
All conditions are the same as the conditions for the direct-indexed-address address-operand type except the indirect addressing mode has been selected.		

Table 5-6. ASSEMBLER EXECUTION OF DIRECT OR INDIRECT INDEXED ADDRESS WITHOUT DISPLACEMENT VALUE

Example:	LD	R1, (R3)
	LD	R1, @ (R3)
In this case, the displacement field is set to zero, and the generated instruction code is absolute. The effective address at execution time is either the contents of the index register or the contents of the memory location pointed to by the index register (indirect addressing).		

Chapter 6

IMP-16 ASSEMBLER PROGRAMS

6.1 DEFINITION

The IMP-16 "cross assembler program" assembles an object program for a source program on a host computer for subsequent execution by an IMP-16 microprocessor. The assembler may be used on different host processors since it is written in FORTRAN IV (USA Standard Language Subset). It requires the following minimum peripheral hardware complement: processor input unit, scratch unit, list output unit, and binary output unit.

The cross assembler accepts free-format source statements and, in two passes, produces an unlinked "relocatable load module" (object program) and a program listing.

The IMP-16 "resident assembler" assembles an object program from a source program on an IMP-16 computer and accepts input for assembly from the card reader, the paper tape reader, or the Teletypewriter keyboard. It is a three-pass assembler that produces an assembler listing on pass-two and a binary-output paper tape on pass-three.

Salient features of the IMP-16 assembler program follow.

- Relocatable or absolute load module generation
- Conditional assembly facilities
- Global symbols for communication between independent programs
- Local symbols
- Wide variety of assembly time operators (+, -, *, /, AND, OR, NOT)
- Diagnostic messages that include error position in source line

Reference 5 describes how to use the IMP-16 cross assembler and the related programs installed on the TYM-SHARE nation-wide timesharing system.

6.2 PROGRAM RELOCATION

The programmer should have an understanding of the relocation mechanism and the way in which the IMP-16 loader acts on the unlinked Relocatable Load Module (RLM) before the assembler is used.

Three types of object codes can be generated by the assembler in the RLM: (1) absolute, (2) base sector relocatable, and (3) top sector relocatable. The type or types of object code that exist in a particular RLM depends on the programmer's use of the "program section directives" (.ASECT, .BSECT, and .TSECT) that enable the programmer to create the program in sections, producing a load module that is absolute, base sector relocatable, top sector relocatable, or a combination of the three. The programmer may control the start of both the base-sector and the top-sector sections. Absolute sectors are always loaded where they were assembled. Therefore, the programmer must exercise some care to ensure that an absolute section does not overlay an existing base- or top-sector section. Usage of the lower 256 memory locations (base sector) should be minimized because they are used for both intra- and inter-RLM linkages and other absolute data such as interrupt processing routines, shared data, debugging routines, and other routines. To conserve memory space, one RLM's base sector (or top sector) immediately follows the base or top sector from the previous RLM unless other direction is given.

A memory map showing the locations of an RLM containing all three sectors is shown in figure 6-1.

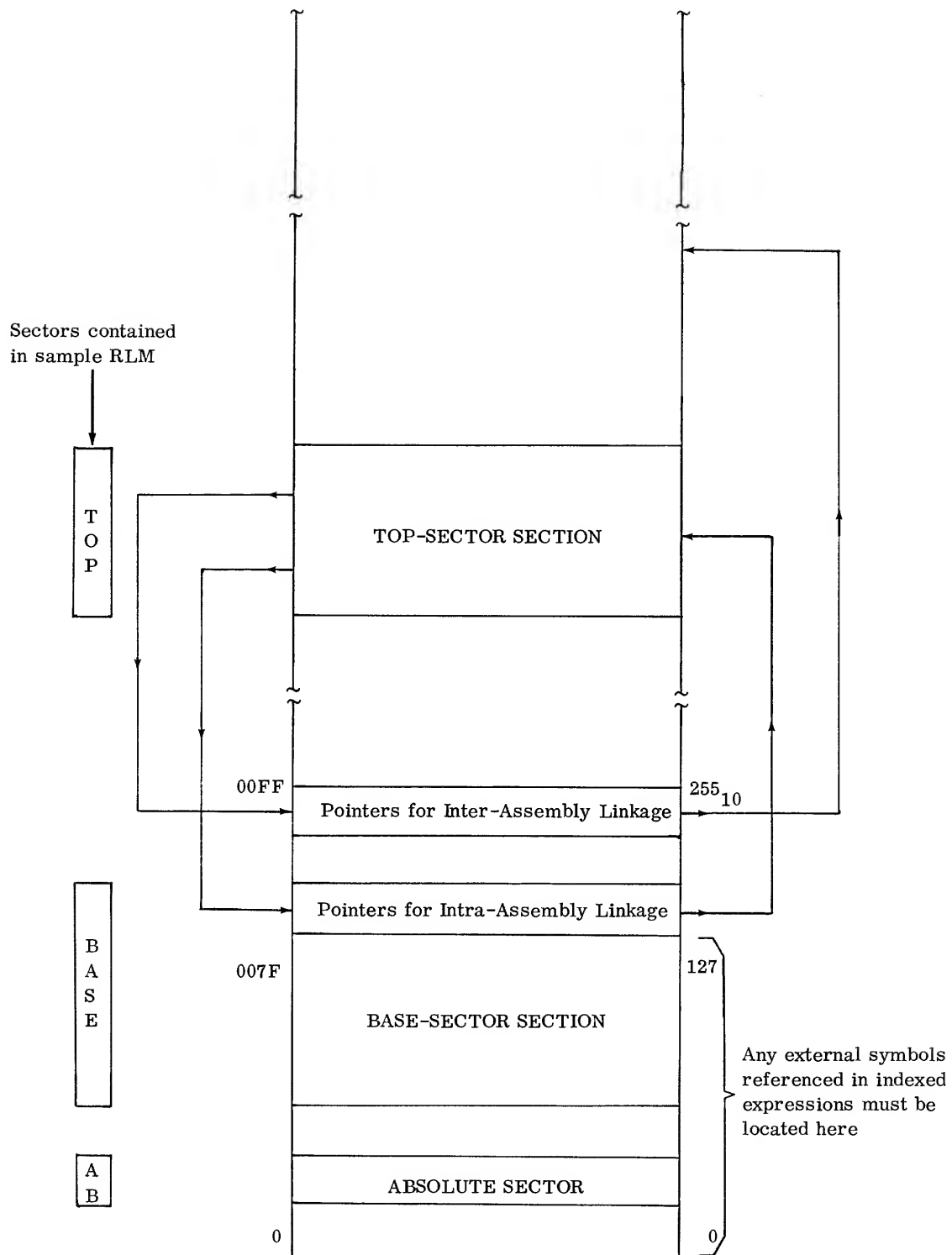


Figure 6-1. Memory Map

6.3 INPUT AND OUTPUT

The input and output files (data sets) required by the cross assembler are listed below:

FORTRAN File Name (DDNAME)	Function	File Format	Logical Record Length
FT05F001	Source File (Input)	Sequential	80 bytes
FT06F001	Listing File (Output)	Sequential	121 bytes
FT09F001	Relocatable Load Module (Out)	Binary	18 words

6.3.1 Source File (Input)

The source file may be input via punched cards, paper tape, or from the keyboard of a computer terminal.

6.3.2 Program Listing File (Output)

The program listing contains ASA-standard carriage control characters. The format of the program listing written from this file follows. An example program listing is shown in figure 2-2.

Each line in the cross assembler program listing contains the following sequential columns: line number, location counter, value, indicator, source statement error message.

Where:

LINE NUMBER — decimal line number of the source input statement. All source statements not deleted by conditional assembly directives are assigned sequential numbers.

LOCATION COUNTER — current hexadecimal value of the location counter. Any labels in the source statements are assigned this value.

VALUE — hexadecimal value of the code generated (or assignment made). For Assembler Statements that do not generate code, this field is blank.

INDICATOR — 1-character symbol that describes the relocation characteristics of the code generated. The symbols are as follows:

<u>Symbol</u>	<u>Interpretation</u>
A	Absolute - value will not be changed.
B	Base Sector Relocation - base sector relocation constant is added.
T	Top Sector Relocation - top sector relocation constant is added.
I	Indirect Address Generated - base sector relocation constant is added.
E	External - instruction address is linked to a symbol that is external to the assembly.
F	Form - word was generated by a .FORM statement.

SOURCE STATEMENT — reproduction of the source statement.

ERROR MESSAGE — will appear on the line(s) following the statement line if an error is detected. The question mark to the right of the error message designates, as closely as possible, the position of the error in the statement.

Error messages are defined in appendix G.

At the end of the program listing, a list of generated pointers is provided (if generated anywhere), a symbol table is produced, a message is printed noting the number of errors discovered by the assembler program, and the source and object checksums are printed.

6.3.3 Relocatable Load Module (Output)

The relocatable load module (RLM) contains the object code produced from the source statements, the relocation information, and the external linkage details. The RLM file is written as an unformatted file.

6.4 TYPES OF RLM RECORDS

The RLM file is composed of a series of records each comprising eighteen 16-bit words. The representation of these records depends on the storage medium. There are four types of RLM records:

- Title Record (one per RLM)
- Symbol Record (variable number per RLM)
- Data Record (variable number per RLM)
- End Record (one per RLM)

An RLM record is punched on cards in hexadecimal characters using four columns per word. Thus, 18 words occupy columns 1 through 72. Each column contains the Hollerith characters for the corresponding hexadecimal digit.

On paper tape, an RLM record is punched by the IMP-16 resident assembler as follows:

- Start of Text Character X'02
- RLM Record (each word occupies two 8-bit frames)
- Carriage Return (OD)
- Line Feed (OA)
- 8 Null Frames

The loaders scan for a "start of text character," then process the record whose length is specified in the first word (see figure 6-2) and then ignore subsequent frames until a new "start of text character" is encountered.

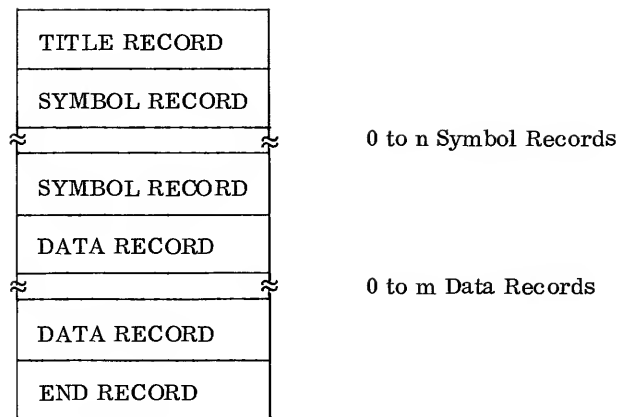
The records are produced in the sequence illustrated in figure 6-2A. Independent of the record type, the first two words (figure 6-2B) in each record always have the same interpretation. The first word specifies the record type and the length of the record body. The second word contains a checksum for error detection.

6.4.1 Title Record

The title record identifies the RLM by name and, optionally, by a qualifying character string. These two items are supplied by the last .TITLE directive statement in the source program. If this directive is not included, a default name (MAINPR) is used. If the default name is assigned, the qualifying character string is empty. Also included in the title record are two values that specify the amount of storage utilized in the base sector and the top sector of memory. The method for determining the storage utilization is by keeping track of the maximum value held by two respective location counters. Figure 6-3 illustrates the format of the title record.

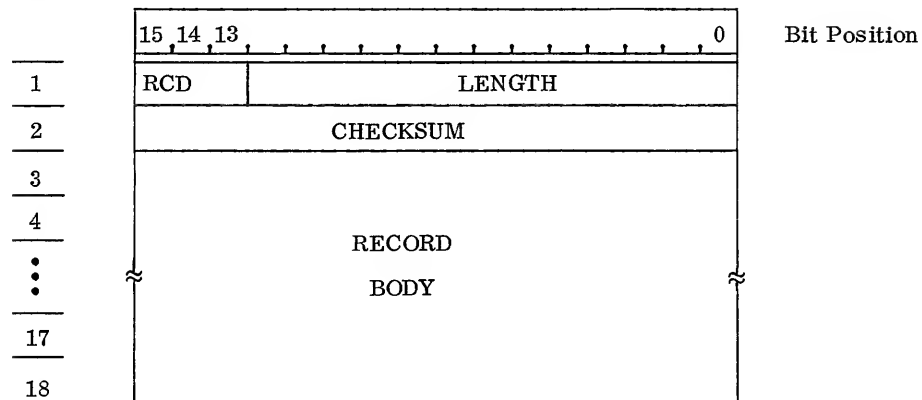
6.4.2 Symbol Records

The symbol records specify values for global symbols that are internal to the current RLM. These symbols can then be referenced by other RLMs. In addition, global symbols that are external to the RLM are specified with associated linkage information. Figure 6-4 illustrates the format of the symbol record.



Record
Word
Number

View A. RLM File Format



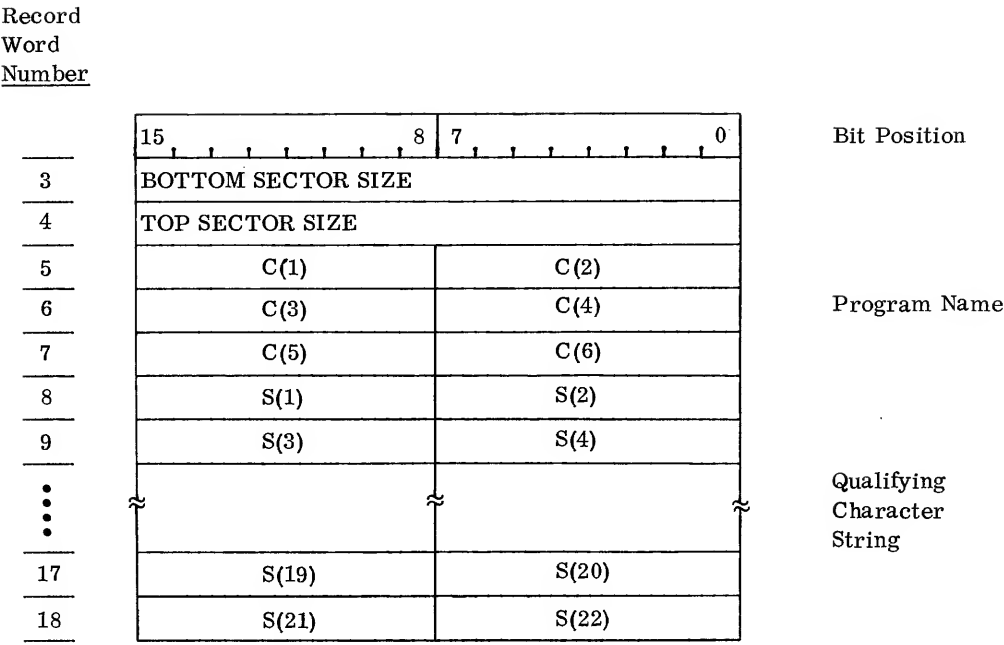
Notes: 1. RCD specifies the type of record

<u>RCD</u>	<u>Record Type</u>
0	Title
1	Symbol
2	Data
3	End

2. The CHECKSUM is formed by taking the arithmetic sum of all the words in the record body.

View B. General Record Format

Figure 6-2. RLM File and General Record Formats



- Notes:
- 1. C(i) and S(i) are 7-bit ASCII characters.
 - 2. If there are more than 22 characters in the qualifying string, only the first 22 are used.

Figure 6-3. Title Record Format

Record
Word
Number

	15	14	13	12	11	10	9	8	7								0	Bit Position
3	TYP 1			TYP 2			TYP 3			NOT USED								
4	SYM1 (1)								SYM1 (2)									
5	SYM1 (3)								SYM1 (4)									
6	SYM1 (5)								SYM1 (6)									
7	VALUE 1																	
8	SYM2 (1)								SYM2 (2)									
9	SYM2 (3)								SYM2 (4)									
10	SYM2 (5)								SYM2 (6)									
11	VALUE 2																	
12	SYM3 (1)								SYM3 (2)									
13	SYM3 (3)								SYM3 (4)									
14	SYM3 (5)								SYM3 (6)									
15	VALUE 3																	
16	NOT USED																	
17																		
18																		

Notes: 1. TYP(i) specifies the relocation mode for symbol i.

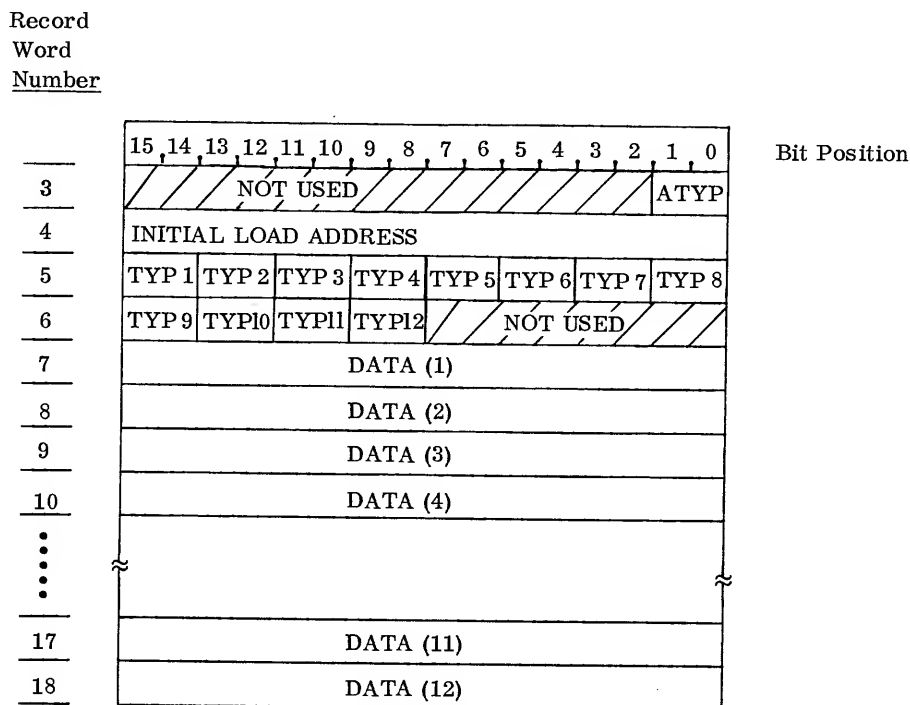
TYP (i)	Relocation Mode
0	absolute
1	base sector
2	top sector
3	external

2. VALUE_i is the absolute address (TYP(i) = 0), relocatable address (TYP(i) = 1 or 2), or external reference number (TYP(i) = 3).
3. SYM_i(j) are 7-bit ANSI characters. If a symbol is less than six characters long, the remaining characters are zero.

Figure 6-4. Symbol Record Format

6.4.3 Data Record

The data records contain the actual data and the instruction words to be loaded into memory. Each data record contains the initial load address and the address mode for the first data word in the record. Subsequent data are loaded sequentially. Also, for each data word, there is a 2-bit field that specifies relocation information. Any time a location or a discontinuity (that is, change of sector or empty block) exists in the data to be loaded, the current record is terminated (possibly with fewer than 12 data words) and a new record is initiated. Figure 6-5 illustrates the data record format.



Notes: 1. ATYP specifies the address mode for the INITIAL LOAD ADDRESS.

ATYP	Address Mode
0	absolute
1	base sector relocatable
2	top sector relocatable

2. TYP(i) specifies the relocation mode for DATA(i).

TYP(i)	Relocation Mode
0	absolute
1	base sector
2	top sector
3	external

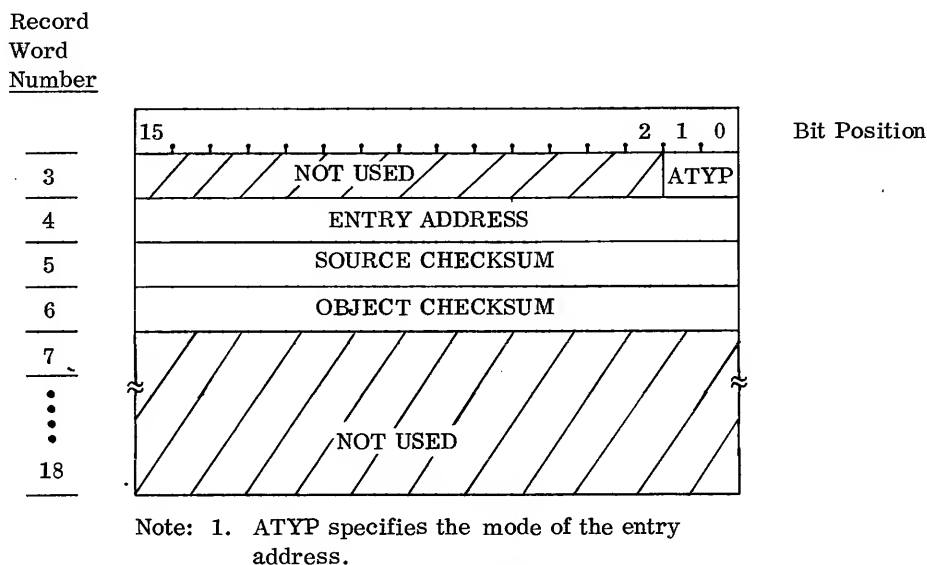
Figure 6-5. Data Record Format

6.4.4 End Record

The end record marks the end of the RLM file and specifies an entry address for the load module. The end record format is illustrated in figure 6-6.

The source checksum represents the sum (modulo- 2^{16}) of all the characters, taken one at a time, in the program source file. This sum is printed on the program listing following the symbol table printout.

The object checksum represents the modulo- 2^{16} sum of all the individual record checksums of the RLM. This sum is also printed on the program listing following the symbol table.



ATYP	Address Mode
0	absolute
1	base sector
2	top sector
3	external

Figure 6-6. End Record Format

6.5 LOADING OBJECT PROGRAM INTO IMP-16

The relocatable load module output by the IMP-16 cross assembler program must be reformatted before it can be loaded into IMP-16 memory for execution. The type of reformatting depends on the loading method used. Three loading methods are available, each of which involves tradeoffs among the complexity of the loading process, the amount of work that must be done by the user, and the flexibility available to the user at load time (vs. assembly time).

Several IMP-16 programs are available for loading reformatted RLMs into memory of the IMP-16 for execution. In addition ANSI FORTRAN programs are available to reformat the RLM output by the assembler program into a format suitable for each loader. The loading methods, the loaders available for each method, and the corresponding reformatting programs are defined below. (A detailed description of each loader is discussed in the appropriate IMP-16 Utilities Reference Manual.)

6.5.1 Bootstrap Loaders

The simplest loading process involves bootstrapping the program into memory. Bootstrap loading consists of using a simple bootstrap loading program (CRBOOT or PTBOOT) to read a program, preconverted to its exact memory format, into a fixed area of main memory for execution.

Bootstrap loading is very rapid, but only one program can be loaded before execution begins. The bootstrap program may be (1) resident in Read Only Memory (ROM), (2) loaded by a hardware function, or (3) loaded by the user via the control panel.

The preconversion of the program to its exact memory format implies the following:

1. Allocation of the entire program to an absolute fixed memory region prior to assembly. This must be done by the programmer and reduces flexibility at load time.
2. Use of a utility program that will reformat the RLM into an IMP-16 bootstrap format.

The following are bootstrap loaders:

1. CRBOOT loads the formatted 72 hexadecimal characters from each card into 18 successive IMP-16 memory locations. When loading is completed, execution begins.
2. PTBOOT loads an 8-channel binary paper tape into memory. The paper tape must be preformatted to contain the initial load address as the first word of the tape, the number of words to be loaded as the second word, and the start address as the last word of the tape. When loading is completed, execution begins.

IMPPCRB is an ANSI FORTRAN program that is available for preparing memory images suitable for loading by CRBOOT. IMPPCRB preloads one or more RLMs into simulated IMP-16 memory (without relocation or inter-module linking) and punches a hexadecimal memory image onto cards in a format suitable for loading by CRBOOT. See figure 6-7.

In order to convert Assembler output to paper tape format, a 2-step process is required. This process is described in 6.5.4.

6.5.2 Absolute Loaders

An absolute loader is used to load one or more programs into preallocated, fixed areas of memory. In order to use this type of loader, the user must decide, before assembly, the exact memory areas to be occupied by each of his programs. Also, any linking of one program to another or to common, shared data must be accomplished at assembly time by assignment of common labels to fixed, absolute addresses in memory. The advantages of this method are that a small, simple loader may be used and that no commands are required at load time. The absolute loader may be resident in ROM (Read Only Memory) or may be loaded by a bootstrap loader.

The following are absolute loaders:

1. ABSCR reads one or more RLMs from the card reader, loads them into specified memory locations, and transfers control to the specified entry point.
2. ABSPT reads one or more RLMs from the paper tape reader, loads them into specified memory locations, and transfers control to the specified entry point.

IMPPRLM is an ANSI FORTRAN program that transcribes RLMs, as output by the IMP-16 assembly program, to punched cards in the format required by ABSCR and GENLDR (described next). The sequence of operations shown in figure 6-8 is necessary to prepare input for these loaders.

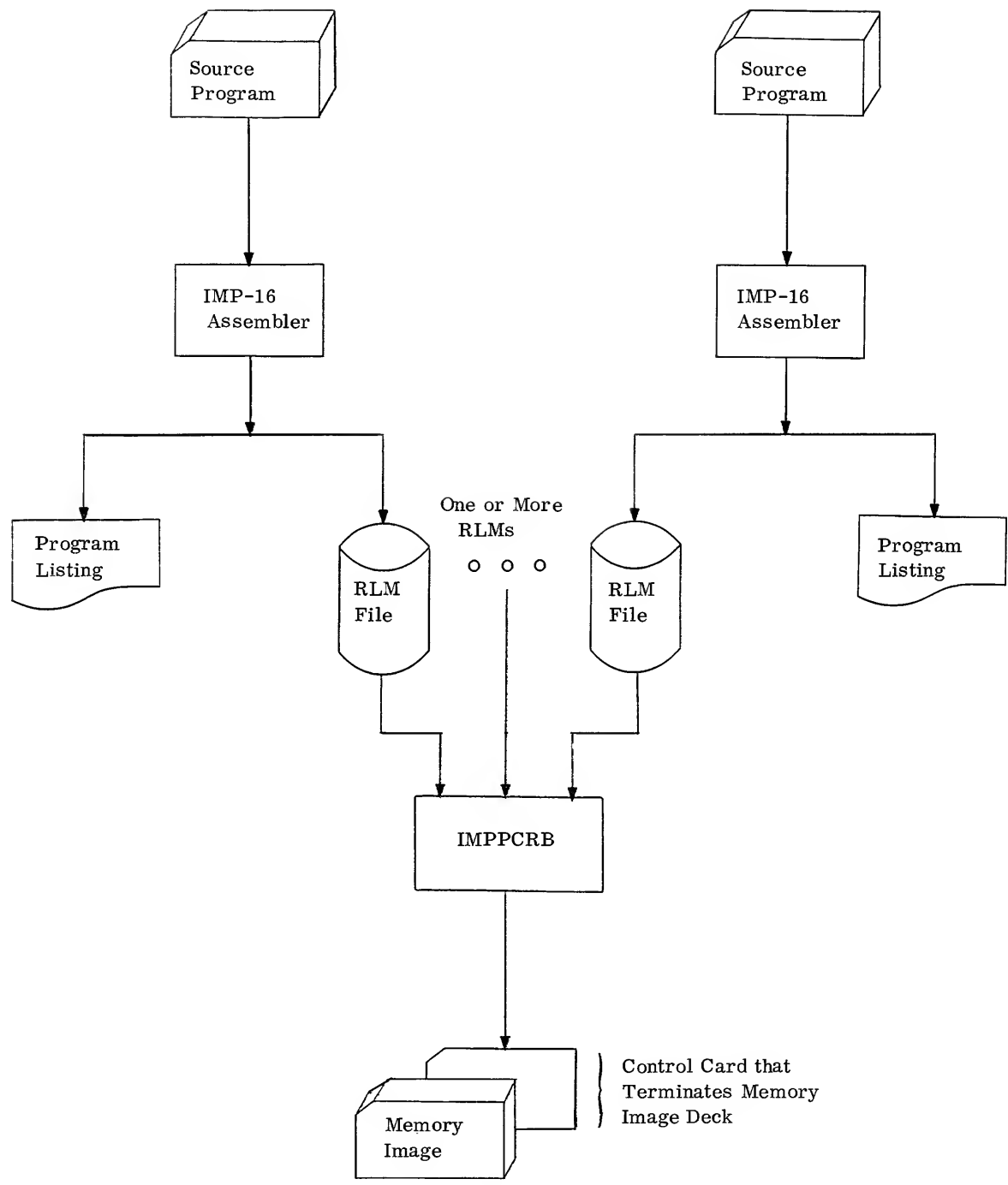


Figure 6-7. Operational Sequence for Preloading and Generating Memory Image Deck for Loading by CRBOOT

Input for ABSPT is prepared with IMPPRLM with one additional step. The cards must be transcribed onto paper tape. Only the first "n" columns of an RLM card are represented on paper tape; "n" is the number of the first blank column, or 73 if columns 1 through 72 are nonblank. This leaves sequence numbers and trailing blanks off the tape. On cards, an object word is represented as four hexadecimal characters; on tape a word is represented by two 8-bit characters. Each paper tape record must be preceded by a STX (Start of Text) character, X'02.

6.5.3 Linking Loader

The most-complex loading process involves relocation of programs and linking among several programs and their shared data at load time. Such a program allows the user to assemble each of his programs relative to memory location zero and will either follow its own method of allocating programs to available memory areas or follow instructions given by the user as to where each program should be loaded. When using this loader, the user may also designate, at assembly time, certain labels within or referenced by his programs as global labels, to be defined or referenced by other programs. At load time, the linking loader connects each reference to a global label (subroutine or data) to its absolute address in memory. This allows the programmer to make all decisions about memory allocation at load time and relieves him of the problem of linking shareable subroutines or data. The disadvantage of this loader is its large size, which precludes usage of a large part of main memory for code or preset data. The linking loader may be loaded by an absolute loader.

The linking loader is GENLDR, a command-driven IMP-16 program that reads one or more RLMs from either the card reader or the paper tape reader, resolves intermodule linkages, relocates object code (as directed at assembly time by .ASECT, .BSECT, or .TSECT directive statements) and transfers control to the specified entry point.

The reformatting program is IMPPRLM described previously.

6.5.4 Reformatting RLM Output on Other Media or in Other Formats

In general, to convert an assembler output to other media or other formats, a 2-step process is required:

1. The RLMs output by the assembler program must be preloaded into a vector in memory-image form.
2. An output routine must be written that will output the generated vector onto the proper media in the format desired.

An ANSI FORTRAN subroutine, FOLD16, may be called by a user program to perform step 1 above. Appendix E contains a program description of FOLD16.

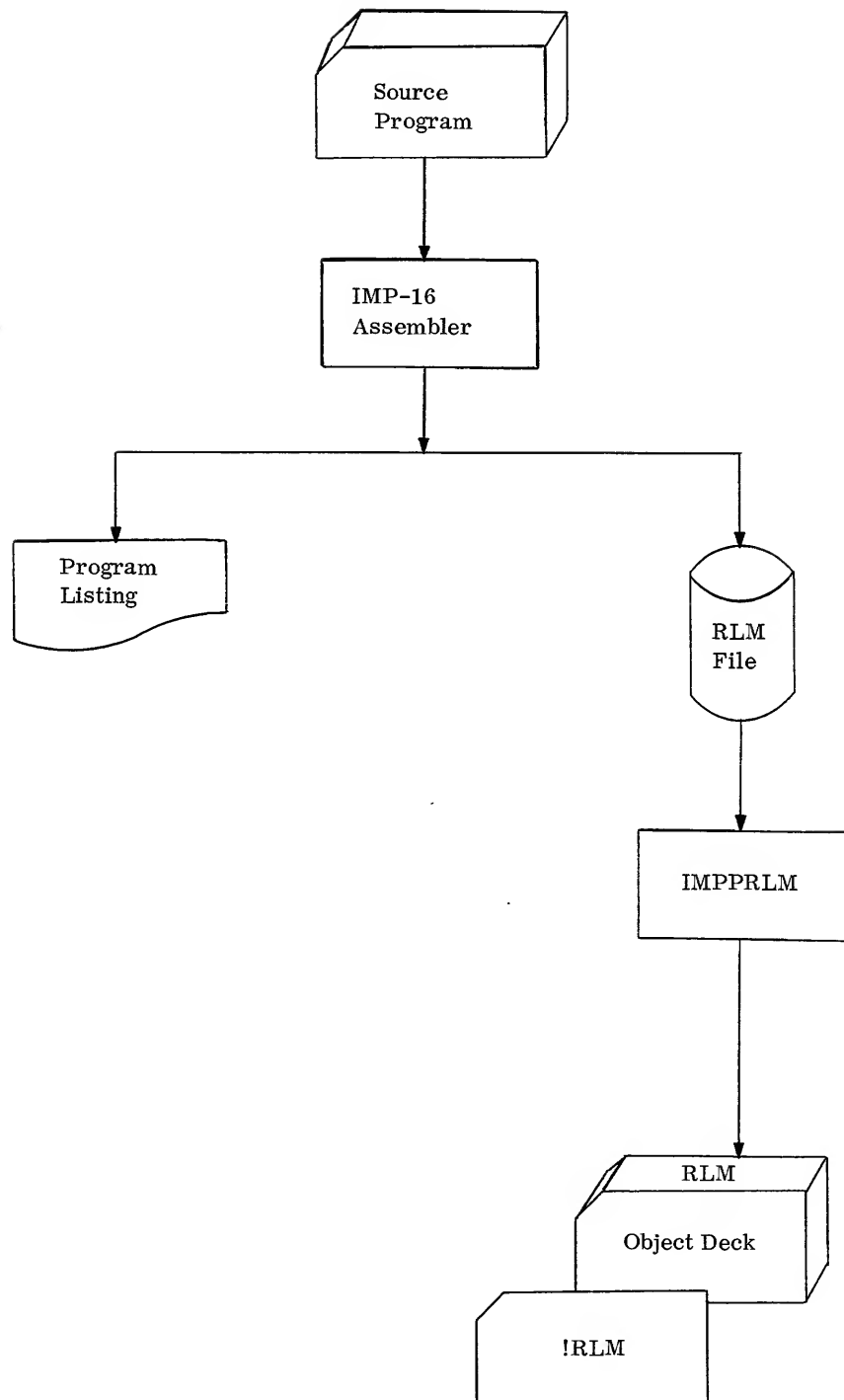


Figure 6-8. Operational Sequence for Preparation of Input for ABSCR or GENLDR

Chapter 7

INSTRUCTION STATEMENTS

7.1 INTRODUCTION

Instruction statements specify machine instructions. An instruction statement is composed of an instruction operator and zero, one, or more operands, depending on the particular instruction. It may be preceded by one or more labels and may be followed by a comment.

There are 61 Instruction Statements. The basic instructions contain 40 instruction statements and the extended instructions contain 21 instruction statements. The extended instructions cannot be executed unless the user has the CROM implementing that instruction set. The instruction statements are summarized in appendix I. Detailed descriptions of the instruction statements are given in this chapter. Instruction statements comprise twelve functional types:

1. Load and Store Instructions
2. Byte Instructions
3. Single-Precision Arithmetic Instructions
4. Double-Precision Arithmetic Instructions
5. Logical Instructions
6. Register Instructions
7. Bit and Status Flag Instructions
8. Transfer-of-Control Instructions
9. Skip Instructions
10. Shift Instructions
11. Interrupt Handling Instructions
12. Input/Output, Halt, and Control Flag Instructions

The following nomenclature is employed in describing the coding structure of the statements.

Capital letters represent literal text that must appear in the coded instructions.

Lowercase letters name an exposition. The field designators used follow:

@	indirect addressing
accumulator	register field (value = 0 or 1)
address	address field
immed	immediate operand (-128 through 127)
immed3	positive 3-bit immediate operand (0-7)
immed4	positive 4-bit immediate operand (0-15)
+immed	positive immediate operand (0 through 127)
register	register field (value 0, 1, 2, or 3)
spaddr	special address field
(xr)	register field (value = 2 or 3)

Brackets around a field indicate that the field is optional.

Refer to table 7-1 for definitions of the notation and symbols used in the operation descriptions of each instruction statement.

The machine language for each instruction is described in the appropriate application or users manual; for example, the IMP-16C Application Manual, the IMP-16L Users Manual, and the IMP-16P Users Manual. See appendix J.

The name of each instruction and its opcode mnemonic (in parentheses) are given as the heading preceding the description of the instruction.

The address class referenced in the notes in each instruction description are defined in 5.4.5.

Table 7-1. Notations and Symbols Used in Operational Descriptions

The notations are listed in alphabetical order. The symbols are listed on the following page. Upper-case mnemonics refer to fields in the instruction word; lower-case mnemonics refer to the numerical value of the corresponding fields. In cases where both lower- and upper-case mnemonics are composed of the same letters, only the lower case mnemonic is given in table 7-1. The use of the lower-case notation designates variables.

Notation	Meaning
ACr	Denotes a specific working register (AC0, AC1, AC2, or AC3), where r is the number of the accumulator referenced in the instruction.
AR	Denotes the address register used for addressing memory or peripheral devices.
cc	Denotes the 4-bit condition code value for conditional branch instructions.
ctl	Denotes the 7-bit control-field value for flag, input/output, and miscellaneous instructions.
CY	Indicates that the Carry flag is set if there is a carry due to the instruction (either an addition or a subtraction).
disp	Stands for displacement value and it represents an operand in a nonmemory reference instruction or an address field in a memory reference instruction. It is a signed two's-complement number except when base page is referenced; in the latter case, it is unsigned.
dr	Denotes the number of a destination working register that is specified in the instruction-word field. The working register is limited to one of four: AC0, AC1, AC2, or AC3.
EA	Denotes the effective address specified by the instruction directly or by indexing. The contents of the effective address are used during execution of an instruction.
fc	Denotes the number of the referenced flag.
INTEN	Denotes the Interrupt Enable control flag.
IOREG	Denotes an input/output register in a peripheral device.
L	Denotes 1-bit link (L) flag.
OV	Indicates that the overflow flag is set if there is an overflow due to the instruction (either an addition or a subtraction).
PC	Denotes the program counter. During address formation, it is incremented by 1 to contain an address 1 greater than that of the instruction being executed.
r	Denotes the number of a working register that is specified in the instruction-word field. The working register is limited to one of four: AC0, AC1, AC2, or AC3.
SEL	Denotes the Select control flag. It is used to select the carry or overflow for output on the carry and overflow (CYOV) line of the CPU, and to include the link bit (L) in shift operations.
sr	Denotes the number of a source working register that is specified in the instruction-word field. The working register is limited to one of four: AC0, AC1, AC2, or AC3.
STK	Denotes the Last-In-First-Out stack in the CPU.

Table 7-1. Notations and Symbols Used in Operational Descriptions (Cont.)

Notation	Meaning
xr	When not zero, this value designates the number of the register to be used in the indexed memory-addressing mode.
()	Denotes the contents of the item within the parentheses. (ACr) is read as "the contents of ACr." (EA) is read as "the contents of EA."
[]	Denotes an optional field in the assembler instruction format.
~	Indicates the logical complement (ones complement) of the value on the right-hand side of ~.
←	Means "is replaced by."
@	Appearing in the operand field of an instruction, denotes indirect addressing.
∧	Denotes an AND operation.
∨	Denotes an OR operation.
⊕	Denotes an exclusive OR operation.

7.2 LOAD AND STORE INSTRUCTIONS (Basic Set)

The instructions that transfer data out of main storage (memory) to a working register (accumulator) for processing or transfer data into memory from an accumulator are the load and store instructions. The load instructions load data into an accumulator, and the store instructions store data into memory.

7.2.1 Load Direct (LD) — Basic Instruction Set

The working register is loaded with the contents of the effective address.

Operation: (ACr) ← (EA)

Coding format: Operation Operand

LD register, address [(XR)]

The operand field contains the address of a working register and an effective address. The address field contains one of the following:

1. An explicit address for base-sector or top-sector addressing
2. A displacement value immediately followed by the address of the base register enclosed in parentheses

Notes: 1. The initial contents of the working register are lost.
 2. The contents of the addressed memory location are unchanged.
 3. Address class 2.

7.2.2 Load Indirect (LD) — Basic Instruction Set

The working register is loaded with the contents of the address specified by the effective address.

Operation: $(ACr) \leftarrow ((EA))$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	LD	register, @address [(XR)]

The operand field contains the address of a working register and an effective address. The address field may contain a base-sector, top-sector, or an indexed address (see 7.2.1).

- Notes:
1. The initial contents of the working register are lost.
 2. The contents of the addressed memory location are unchanged.
 3. The contents of the index register are unchanged.
 4. Address class 2.

7.2.3 Store Direct (ST) — Basic Instruction Set

The contents of the working register are stored in the memory location specified by the effective address.

Operation: $(EA) \leftarrow (ACr)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ST	register, address [(XR)]

The operand field contains the address of a working register and an effective address. The effective address contains one of the following:

1. Contains an explicit address for base-sector or top-sector addressing.
2. Contains a displacement value immediately followed by the address of the base register enclosed in parentheses.

- Notes:
1. The initial contents in the addressed memory location are lost.
 2. The contents of the working register are unchanged.
 3. Address class 2.

7.2.4 Store Indirect (ST) — Basic Instruction Set

The contents of the working register are stored in the memory location specified by the contents of the effective address.

Operation: $((EA)) \leftarrow (ACr)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ST	@register, address [(XR)]

The operand field contains the address of a working register and an effective address. The address field may contain a base-sector, top-sector, or indexed address (see 7.2.4).

- Notes:
1. The initial contents in the addressed memory location are lost.
 2. The contents of the working register are unchanged.
 3. The contents of the index register are unchanged.
 4. Address class 2.

7.3 BYTE INSTRUCTIONS (Extended Set)

The six byte instructions allow the user to load or store an 8-bit byte as opposed to the 16-bit word loaded or stored by the load and store instructions. The byte instructions simplify character manipulation.

7.3.1 Load Byte (LDB) — Extended Instruction Set

The load byte instruction loads the low-order byte of AC0 with a byte from the $EA \div 2$. If the low-order bit of the effective address is 1, the low-order byte is loaded; otherwise, the high-order byte is loaded. The addressing range for this instruction is 0 through $7FFF_{16}$.

Operation: Low-order byte of (AC0) \leftarrow byte from $(EA \div 2)$; SEL \leftarrow 0

Coding Format:	<u>Operation</u>	<u>Operand</u>
	LDB	address $[(XR)]$

The operand field specifies the direct address.

- Notes:
1. $EA \div 2$ is the effective address shifted right one position.
 2. The high-order byte of AC0 is set equal to zero.
 3. The select flag is cleared.
 4. Load Byte uses a double-word machine instruction format.
 5. Load Byte is an extended instruction.
 6. PC-relative addressing is not recommended and is not allowed by the assembler.
 7. Address class 4.

7.3.2 Load Left Byte (LLB) — Extended Instruction Set

The load left byte instruction loads the low-order byte of AC0 with the high-order byte of $EA \div 2$. The load left byte instruction forces the low-order bit of EA to 0, ensuring that the high-order byte is loaded.

The operation, coding format, and notes are identical to those for the load byte instruction (7.3.1).

7.3.3 Load Right Byte (LRB) — Extended Instruction Set

The load right byte instruction loads the low-order byte of AC0 with the low-order byte of $EA \div 2$. The load right byte instruction forces the low-order bit of EA to 1, ensuring that the low-order byte is loaded.

The operation, coding format, and notes are identical to those for the load byte instruction (7.3.1).

7.3.4 Store Byte (STB) — Extended Instruction Set

The store byte instruction stores the low-order byte of AC0 into the byte of $EA \div 2$ as specified by the low-order bit of the effective address. If the low-order bit is 1, the low-order byte is specified; otherwise, the high-order byte is specified. The addressing range for this instruction is 0 through $7FFF_{16}$.

Operation: Byte of $(EA \div 2) \leftarrow$ low-order byte from (AC0); $SEL \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	STB	address $[(XR)]$

The operand field specifies the direct address.

- Notes:
1. $EA \div 2$ is the effective address shifted right one position.
 2. The unspecified byte of $EA \div 2$ and the contents of AC0 are unaffected.
 3. The select flag is cleared.
 4. STB uses a double-word machine instruction format.
 5. STB is an extended instruction.
 6. PC-relative addressing is not recommended and is not allowed by the assembler.
 7. Address class 4.

7.3.5 Store Left Byte (SLB) — Extended Instruction Set

The store left byte instruction stores the low-order byte of AC0 into the high-order byte of $EA \div 2$. The store left byte instruction forces the low-order bit of EA to 0, ensuring that the byte is stored in the high order-byte of $EA \div 2$.

The operation, coding format, and notes are identical to those for the store byte instruction (7.3.4).

7.3.6 Store Right Byte (SRB) — Extended Instruction Set

The store right byte instruction stores the low-order byte of AC0 into the low-order byte of $EA \div 2$. The store right byte instruction forces the low-order bit of EA to 1, ensuring that the byte is stored in the low-order byte of $EA \div 2$.

The operation, coding format, and notes are identical to those for the store byte instruction (7.3.4).

7.4 SINGLE-PRECISION ARITHMETIC INSTRUCTIONS (Basic and Extended Sets)

The four single-precision arithmetic instructions effect algebraic addition, subtraction, multiplication, and division of 16-bit binary operands. Add and subtract are single-word instructions and multiply and divide are double-word instructions.

The carry (CY), overflow (OV), and link (L) bits in the status register are automatically set or reset depending on the result of an arithmetic operation. The OV bit is set whenever an add or subtract operation causes a carry out of bit 14 different from that out of bit 15; otherwise it is a 0. The CY bit is set whenever an add or subtract operation causes a carry out of bit 15; otherwise it is a 0. The L bit is used for arithmetic operations (multiply and divide), and optionally as described later, in shift and rotate operations (7.11). Therefore, activation is controlled by the SEL control flag. If the SEL flag is set, L is activated. If the SEL flag is 0, L is deactivated. If the L bit is activated, it is treated as the high-order bit of a 17-bit register, formed by linking the L bit and an accumulator. A detailed explanation of the status bits used in arithmetic operations is given in appendix B.

7.4.1 Add (ADD) — Basic Instruction Set

The contents of the working register are added algebraically to the contents of the effective address, and the sum is stored in the working register.

Operation: $(ACr) \leftarrow (ACr) + (EA), OV, CY$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ADD	register, address $[(XR)]$

The operand field contains the address of the working register and a direct effective address, specifying the augend.

- Notes:
1. The contents of the addressed memory location are unchanged.
 2. The initial contents of the working register are lost.
 3. The carry and overflow flags are set according to the result of the operation.
 4. Add is a basic instruction.
 5. Address class 1.

7.4.2 Subtract (SUB) — Basic Instruction Set

The contents of the working register are added to the twos complement of the contents of the effective address. The result is stored in the working register.

Operation: $(ACr) \leftarrow (ACr) + \sim (EA) + 1, OV, CY$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SUB	register, address $[(XR)]$

The operand field contains the address of a working register and a direct effective address, specifying the subtrahend.

- Notes:
1. The contents of the addressed memory location are unchanged.
 2. The initial contents of the working register are lost.
 3. The carry and overflow flags are set according to the result of the (twos complement) operation.
 4. Subtract is a basic instruction.
 5. Address class 1.

7.4.3 Multiply (MPY) — Extended Instruction Set

The multiply instruction automatically uses the first (AC0) and second (AC1) working registers. It is the programmer's responsibility to store data from these registers before coding a multiply instruction.

The unsigned integer in the second working register (AC1) is multiplied by the positive integer in the effective address. The high-order part of the 32-bit result is stored in AC0 and the low-order part is stored in AC1.

Operation: $(AC0), (AC1) \leftarrow (AC1) * (EA); SEL \leftarrow 0; L$ altered

Coding Format:	<u>Operation</u>	<u>Operand</u>
	MPY	address $[(XR)]$

The operand field specifies the direct effective address of the memory location containing the multiplier.

- Notes:
1. The previous contents of AC0 and AC1 are lost.
 2. The contents of the addressed memory location are unchanged.
 3. The select flag is cleared.
 4. The link flag is left in an arbitrary state.
 5. MPY uses a double-word machine instruction format.
 6. MPY is an extended instruction.
 7. Address class 3.

7.4.4 Divide (DIV) — Extended Instruction Set

The divide instruction automatically uses the first (AC0) and second (AC1) working registers. It is the programmer's responsibility to store data from these registers before coding a divide instruction.

The positive 32-bit integer in AC0 (high-order part) and in AC1 (low-order part) is divided by the contents of the effective address. The divisor must be a positive number. The integer quotient is placed in AC1 and the remainder in AC0.

Operation: $(AC0), (AC1) \leftarrow (AC0), (AC1) \div (EA); OV; SEL \leftarrow 0; L$ altered

Coding Format:	<u>Operation</u>	<u>Operand</u>
	DIV	address $[(XR)]$

The operand field specifies the direct effective address of the memory location containing the divisor.

- Notes:
1. The overflow flag is set if either of the following results occur:
 - a. The high-order part of the dividend (initial contents of AC0) is greater than or equal to the divisor.
 - b. The quotient is negative.
 2. The select flag is cleared.
 3. The link flag is left in an arbitrary state.
 4. The contents of the addressed memory location are unchanged.
 5. Division by zero is illegal and falls into note 1 (above).
 6. DIV uses a double-word machine instruction format.
 7. DIV is an extended instruction.
 8. Address class 3.

7.5 DOUBLE-PRECISION ARITHMETIC INSTRUCTIONS (Extended Set)

The two-double-precision arithmetic instructions permit algebraic addition and subtraction of 32-bit operands. Double-precision add and double-precision subtract are both double-word instructions.

7.5.1 Double-Precision Add (DADD) — Extended Instruction Set

The double-precision add instruction automatically uses the first (AC0) and second (AC1) working registers. It is the programmer's responsibility to store data from or load data into these registers before coding a DADD instruction.

The double-precision twos-complement value in AC0 (high order) and AC1 (low order) is added to the double-precision twos-complement value in the effective address (high order) and the effective address +1 (low order). The result is stored in AC0 and AC1.

Operation: $(AC0), (AC1) \leftarrow (AC0), (AC1) + (EA), (EA+1); OV; CY, SEL \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	DADD	address $[(XR)]$

The operand field specifies the direct effective address containing the high-order augend.

- Notes:
1. The contents of the effective address and effective address +1 are unchanged.
 2. The overflow or carry flag is set if an overflow or carry occurs; otherwise, they are cleared.
 3. The select flag is cleared.
 4. DADD uses a double-word machine instruction format.
 5. DADD is an extended instruction.
 6. Address class 3.

7.5.2 Double-Precision Subtract (DSUB) — Extended Instruction Set

The double-precision subtract instruction automatically uses the first (AC0) and second (AC1) working registers. It is the programmer's responsibility to store data from or load data into these registers before coding a DSUB instruction.

The double-precision twos-complement value in the effective address (high order) and effective address +1 (low order) is subtracted from the double-precision twos-complement value in AC0 (high order) and AC1 (low order). The result is stored in AC0 and AC1.

Operation: $(AC0), (AC1) \leftarrow (AC0), (AC1) + \sim[(EA), (EA+1)] + 1; OV; CY; SEL \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	DSUB	address $[(XR)]$

The operand field specifies the direct effective address containing the high-order subtrahend.

- Notes:
1. The contents of the effective address and effective address +1 are unchanged.
 2. The overflow or carry flag is set if an overflow or carry occurs; otherwise, they are cleared.
 3. The select flag is cleared.
 4. DSUB uses a double-word machine instruction format.
 5. DSUB is an extended instruction.
 6. Address class 3.

7.6 LOGICAL INSTRUCTIONS (Basic Set)

Logical instructions are used to manipulate data; for example, setting programmable switches used in branch decisions. The operands are treated as 16-bit words. The logical instructions manipulate specified bits in the word. The two instructions, AND or OR, must utilize working registers AC0 or AC1.

7.6.1 Logical AND (AND) — Basic Instruction Set

The logical AND instruction is used to test or set the value of a bit or bits in a word. The working register contains the word with bits to be tested or set. The effective address specifies the data word containing the test or set bits.

At execution time, the two operands are ANDed, and the result is placed in the working register. The AND places a 1 in bit positions in which both operand bits were 1 and places a 0 in bit positions where either operand was a 0. Therefore, zero bits in the memory word, result in zero bits in the word in the register after execution. One bits in the memory word result in a 1 only if there is a 1 in that bit in the register. For example:

Register	0111011011000011
Memory Word	<u>1100110010000001</u>
Register	0100100100000001

Comparison instructions (for example, branch on condition, BOC) are used on the result if data is to be tested.

Operation: $(ACr) \leftarrow (ACr) \wedge (EA)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	AND	accumulator, address $[(XR)]$

The operand field specifies both:

1. The working register containing data. Register must equate to AC0 or AC1.
2. The effective address of data word or mask used to test or set bits.

Notes: 1. The initial contents of the working register are lost.
 2. The contents of the effective address are unchanged.
 3. AND is a basic instruction.
 4. Address class 1.

7.6.2 Logical OR (OR) — Basic Instruction Set

The logical OR instruction is used to insert "1" bits into a word, allowing the programmer to modify part of a word. The working register contains the word to be modified. The address specifies a data word or mask with the appropriate bit(s) set to one.

At execution time the two operands are ORed, and the result is placed in the working register. The OR places a 1 in bit positions in which either operand bit is a 1 and places a 0 where both operand bits are 0's. For example,

Register	1010101001111110
Memory Word	<u>1001110010000100</u>
Register	1011111011111110

Operation: $(ACr) \leftarrow (ACr) \vee (EA)$

Coding Format: Operation Operand
 OR accumulator, address[(XR)]

The operand field specifies:

1. The working register containing the data word. The register must equate to AC0 or AC1.
2. The effective address of the data word of mask used to set bit(s).

Notes: 1. The initial contents of the working register are lost.
 2. The contents of the effective address are unchanged.
 3. OR is a basic instruction.
 4. Address class 1.

7.7 REGISTER INSTRUCTIONS (Basic Set)

The eleven register instructions allow operations between registers, operations on data in a single register, or transfer of data between one register and the stack. The following are register instructions.

Push onto Stack -----	PUSH
Pull from Stack -----	PULL
Exchange Register and Stack -----	XCHRS
Load Immediate -----	LI
Add Immediate, Skip if Zero -----	AISZ
Complement and Add Immediate -----	CAI
Register Add -----	RADD
Register Exchange -----	RXCH
Register Copy -----	RCPY
Register Exclusive-OR -----	RXOR
Register AND -----	RAND

7.7.1 Push onto Stack (PUSH) — Basic Instruction Set.

This instruction stores the contents of the specified register at the top of the stack. The contents of all other levels in the stack are moved down one level. If the stack is full before the push occurs, the contents of the lowest level are lost.

Operation: $(STK) \leftarrow (ACr)$

Coding Format: Operation Operand
 PUSH register

The operand field specifies the address of the register whose contents are pushed to the stack. The register address may equate to AC0, 1, 2, or 3.

Notes: 1. The initial contents of the register are unaltered.
 2. PUSH is a basic instruction.

7.7.2 Pull from Stack (PULL) — Basic Instruction Set

This instruction loads the contents from the top of the stack into the selected register. The contents of each level of the stack move up one level. Zeros enter the bottom of the stack.

Operation: $(ACr) \leftarrow (STK)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	PULL	register

The operand field specifies the address of the register into which the contents of the top level of the stack are loaded. The register address must equate to AC0, 1, 2, or 3.

Notes: 1. The initial contents of the register are lost.
2. PULL is a basic instruction.

7.7.3 Exchange Register and Stack (XCHRS) — Basic Instruction Set

This instruction exchanges the contents of the top level of the stack and the selected register.

Operation: $(STK) \leftarrow (ACr), (ACr) \leftarrow (STK)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	XCHRS	register

The operand field specifies the address of the register in the exchange. The register address must equate to AC0, 1, 2, or 3.

Note: 1. XCHRS is a basic instruction.

7.7.4 Load Immediate (LI) — Basic Instruction Set

In this single register instruction, the value of the immediate operand loaded into the selected register. The value with the sign bit extended through bit 15 replaces the contents of the register. This instruction is often used to set up AC2 and AC3 for indexed addressing.

Operation: $(ACr) \leftarrow \text{disp (sign extended)}$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	LI	register, immed

The operand field specifies the address of the register and contains an immediate operand. The value of the immediate operand is -128 to +127.

Notes: 1. The initial contents of the register are lost.
2. LI is a basic instruction.

7.7.5 Add Immediate, Skip if Zero (AISZ) — Basic Instruction Set

In this single-register instruction, the contents of the selected register are replaced by the sum of the contents of the register and the immediate value (sign bit 7 extended through bit 15). If the new contents of the register equal zero, the contents of the PC are incremented by one, thus skipping the next instruction.

Operation: $(ACr) \leftarrow (ACr) + \text{disp (sign extended)}$, OV, CY
 If new $(ACr) = 0$, $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	AISZ	register, immed

The operand field specifies the address of the register used in the instruction and an immediate operand. The value of the immediate operand is -128 to +127.

- Notes:
1. The initial contents of the register are lost.
 2. The overflow and carry flags are set according to the result of the operation.
 3. If the condition is met, this instruction will cause a skip of ONE word.
 4. AISZ is a basic instruction.

7.7.6 Complement and Add Immediate (CAI) — Basic Instruction Set

In this single-register instruction, the contents of the selected register are complemented and then added to the value of the immediate operand (sign bit extended through bit 15). The result is then stored in the register.

Operation: $(ACr) \leftarrow \sim(ACr) + \text{disp (sign extended)}$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	CAI	register, immed

The operand field specifies the address of the register used in the instruction and an immediate operand. The value of the immediate operand is -128 to +127.

- Notes:
1. The initial contents of the register are lost.
 2. The carry and overflow flags are not affected by this instruction.
 3. Specification of an immediate operand of zero (0) results in a ones complement; an immediate operand of one (1) results in a twos complement.
 4. CAI is a basic instruction.

7.7.7 Register Add (RADD) — Basic Instruction Set

This register-to-register instruction replaces the contents of the destination register with the sum of the contents of the destination register (dr) and the source register (sr).

Operation: $(ACdr) \leftarrow (ACsr) + (ACdr)$, OV, CY

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RADD	source register, destination register

The operand field contains the address of the source register and the address of the destination register. The register addresses must equate to AC0, 1, 2, or 3.

- Notes:
1. The initial contents of the destination register are lost.
 2. The contents of the source register are unaltered.
 3. The overflow and carry flags are set according to the result of the operation.
 4. RADD is a basic instruction.

7.7.8 Register Exchange (RXCH) — Basic Instruction Set

This register-to-register instruction exchanges the contents of the source register (sr) and the destination register (dr).

Operation: $(ACsr) \leftarrow (ACdr)$, $(ACdr) \leftarrow (ACsr)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RXCH	source register, destination register

The operand field contains the address of the source register and the address of the destination register. The register addresses must equate to AC0, 1, 2, or 3.

7.7.9 Register Copy (RCPY) — Basic Instruction Set

This register-to-register instruction replaces the contents of the destination register (dr) with the contents of the source register (sr).

Operation: $(ACdr) \leftarrow (ACsr)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RCPY	source register, destination register

The operand field contains the address of the source register and the address of the destination register. The register addresses must equate to AC0, 1, 2, or 3.

- Notes:
1. The initial contents of the destination register are lost.
 2. The initial contents of the source register are unchanged.
 3. RCPY is a basic instruction.

7.7.10 Register Exclusive OR (RXOR) — Basic Instruction Set

This register-to-register instruction replaces the contents of the destination register (dr) by exclusively ORing the contents of the destination register with the contents of the source register (sr). The exclusive-OR operation excludes the condition in which both operands have a 1 bit; that is, a 1 bit in the first or second operand, but not both, yields a 1 bit in the result. Exclusive-OR is frequently used to test for status or to zero an accumulator by exclusive ORing the accumulator with itself.

Operation: $(ACdr) \leftarrow (ACdr) \vee (ACsr)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RXOR	source register, destination register

The operand field contains the address of the source register and the address of the destination register. The register addresses must equate to AC0, 1, 2, or 3. For example,

Source Register	1010101011111100
Destination Register	<u>1001110010000100</u>
Destination Register	0011011001111000

- Notes:
1. The initial contents of the destination register are lost.
 2. The initial contents of the source register are unchanged.
 3. RXOR is a basic instruction.

7.7.11 Register AND (RAND) — Basic Instruction Set

This register-to-register instruction replaces the contents of the destination register (dr) by ANDing the contents of the destination register with the contents of the source register (sr).

Operation: $(ACdr) \leftarrow (ACdr) \wedge (ACsr)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RAND	source register, destination register

The operand field contains the address of the source register and the address of the destination register. The register addresses must equate to AC0, 1, 2, or 3. For example,

Source Register	0111011011000011
Destination Register	<u>1100110010000001</u>
Destination Register	0100010010000001

- Notes:
1. The initial contents of the destination register are lost.
 2. The initial contents of the source register are unchanged.
 3. RAND is a basic instruction.

7.8 BIT AND STATUS FLAG INSTRUCTIONS (Extended and Basic Sets)

The seven bit and status flag instructions are used to program status flags in the status register or bits in a working register (AC0). The status register is a 16-bit register where the 3 high-order bits automatically reflect the status of arithmetic operations, and the 13 remaining bits, General Purpose flags (GF), are specified and used by the programmer. See figure 7-1 for the arrangement of status flags in the status register.

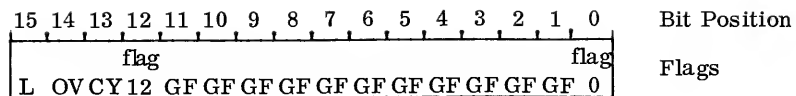


Figure 7-1. Configuration of Status Register

The definitions for the status flags of two IMP-16 microprocessors, the IMP-16C and IMP-16L, are given in table 7-2.

Table 7-2. Definitions of IMP-16C/L Flags

Bit Position	IMP-16C		IMP-16L	
	Name of Flag	Flag Mnemonic	Flag Mnemonic	Name of Flag
15	Link	L	L	Link
14	Overflow	OV	OV	Overflow
13	Carry	CY	CY	Carry
12	General Purpose Flag	GF	IEN3	Interrupt Enable Flag Level 3
11	General Purpose Flag	GF	GF	General Purpose Flag
10	General Purpose Flag	GF	GF	General Purpose Flag
9	General Purpose Flag	GF	GF	General Purpose Flag
8	General Purpose Flag	GF	IEN2	Interrupt Enable Flag Level 2
7	General Purpose Flag	GF	GF	General Purpose Flag
6	General Purpose Flag	GF	GF	General Purpose Flag
5	General Purpose Flag	GF	GF	General Purpose Flag
4	General Purpose Flag	GF	IEN1	Interrupt Enable Flag Level 1
3	General Purpose Flag	GF	GF	General Purpose Flag
2	General Purpose Flag	GF	CF	Reserved for use by control panel
1	General Purpose Flag	GF	CP	Reserved for use by control panel
0	General Purpose Flag	GF	IEN0	Interrupt Enable Flag Level 0

The CY, OV and L bits are automatically set or reset depending on the result of register operations. The thirteen general purpose bits are set or reset by program instruction.

Apart from automatic setting/resetting of the CY, OV, and L bits, testing and resetting of status bits may take place directly in the status register (SETST or CLRST) or in a working register. Testing or setting of bits must take place in a working register. If the operation takes place in a working register, the normal sequence of operations follows:

1. Transfer status or data to register
2. Test/set/reset bits
3. Return accumulator contents to status register or data word

Note that all transfers between the status register and the working registers are via the stack.

In much the same way that the status of the link bit determines whether or not an accumulator is incremented during multiplication, the general-purpose flags may be used to store condition results and control subsequent program branching.

7.8.1 Push Status Flags onto Stack (PUSHF) — Basic Instruction Set

The contents of the top of the stack are replaced by the contents of the status flags in the status register. The previous contents of the top of the stack and lower levels are pushed down one level. The contents of the lowest level of the stack are lost.

Operation: $(STK) \leftarrow (STATUS\ FLAGS)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	PUSHF	not used

Note: PUSHF is a basic instruction.

7.8.2 Pull Status Flags from Stack (PULLF) — Basic Instruction Set

The contents of the Status Register are replaced by the contents of the top of the stack. The previous contents of lower levels of the stack are pulled up by one level with zeros filling the contents of the lowest level.

Operation: $(STATUS\ FLAGS) \leftarrow (STK)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	PULLF	not used

Note: PULLF is a basic instruction.

7.8.3 Set Status Flag (SETST) — Extended Instruction Set

Bit n of the Status Register is set (to a "1" bit). All other bits are unaffected.

Operation: $Status\ Flag\ n \leftarrow 1; SEL \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SETST	immed4

The operand field contains a single numeric ($0 \leq n \leq 15$) that identifies the bit set.

- Notes:
1. The select flag is cleared.
 2. SETST is an extended instruction.

7.8.4 Clear Status Flag (CLRST) — Extended Instruction Set

Bit n of the Status Register is cleared (to a "0" bit). All other bits are unaffected.

Operation: Status Flag $n \leftarrow 0$; SEL $\leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	CLRST	immed4

The operand field contains a single numeric ($0 \leq n \leq 15$) that identifies the bit cleared.

- Notes: 1. The select flag is cleared.
2. CLRST is an extended instruction.

7.8.5 Set Bit (SETBIT) — Extended Instruction Set

Bit n of AC0 is set (1). All other bits are unaffected.

Operation: AC0 _{n} $\leftarrow 1$, SEL $\leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SETBIT	immed4

The operand field contains a single numeric ($0 \leq n \leq 15$) that identifies the bit set.

- Notes: 1. The select flag is cleared.
2. SETBIT is an extended instruction.

7.8.6 Clear Bit (CLRBIT) — Extended Instruction Set

Bit n of AC0 is cleared (to a "0" bit). All other bits are unaffected.

Operation: AC0 _{n} $\leftarrow 0$; SEL $\leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	CLRBIT	immed4

The operand field contains a single numeric ($0 \leq n \leq 15$) that identifies the bit cleared.

- Notes: 1. The select flag is cleared.
2. CLRBIT is an extended instruction.

7.8.7 Complement Bit (CMPBIT) — Extended Instruction Set

Bit n of AC0 is complemented. All other bits are unaffected.

Operation: AC0 _{n} $\leftarrow \sim AC0_n$; SEL $\leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	CMPBIT	immed4

The operand field contains a single numeric ($0 \leq n \leq 15$) that identifies the bit complemented.

- Notes: 1. The select flag is cleared.
2. CMPBIT is an extended instruction.

7.9 TRANSFER-OF-CONTROL INSTRUCTIONS

Instructions are normally executed in sequential order according to the memory locations in which they are stored. The program counter (PC) holds the address of the next instruction to be executed. It is incremented by one, immediately following the fetching of each instruction, during execution of the current instruction.

The programmer uses a transfer-of-control instruction to break the normal sequential execution. When transfer-of-control occurs, the address specified by the instruction replaces the current address in the PC.

There are four types of transfers:

Unconditional Jump - This transfer simply causes program execution to continue at the address specified by the instruction. Jump instructions specify addresses in either direction; for example, before the jump instruction or after the jump instruction in the execution sequence. There are three unconditional jump instructions: Jump (JMP), Jump Indirect (JMP), and Jump Through Pointer (JMPP).

Subroutine Jump - Any frequently used set of instructions may be coded once and used as a subroutine. Subroutines are executed using jump-to-subroutine instructions and the Return-from-Subroutine (RTS) instruction.

Interrupt Jump - Program execution may be interrupted unexpectedly (for example, an incoming instrument reading). Upon sensing an interrupt, the microprocessor hardware forces a jump-to-interrupt-service routine. The Return from Interrupt (RTI) instruction allows execution to return from the interrupt to the point in the main program where the interrupt occurred.

Branch-on-Condition - If a specified condition is true, program execution is transferred to another location; otherwise, the next instruction in consecutive order is executed.

There are ten Transfer-of-Control Instructions:

```

Jump Direct-----JMP
Jump Indirect -----JMP
Jump Through Pointer-----JMPP
Jump to Subroutine Direct -----JSR
Jump to Subroutine Indirect -----JSR
Jump to Subroutine Implied -----JSRI
Jump to Subroutine Through Pointer ----JSRP
Return from Subroutine-----RTS
Return from Interrupt -----RTI
Branch on Condition-----BOC

```

7.9.1 Jump Direct (JMP) — Basic Instruction Set

The effective address replaces the contents of the PC. The next instruction is fetched from the location designated by the new contents of the PC.

Operation: (PC) ← EA

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JMP	address [(XR)]

The operand field contains a base-page or PC-relative address.

- Notes:
1. The initial contents of the PC are lost.
 2. JMP is a basic instruction.
 3. Address class 2.

7.9.2 . Jump Indirect (JMP) — Basic Instruction Set

The contents of the effective address replace the contents of the PC. The next instruction is fetched from the location designated by the new contents of the PC. The @ symbol must be placed in the operand field, not in the operation field.

Operation: $(PC) \leftarrow (EA)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JMP	@address [(XR)]

The operand field contains a base-page or PC-relative address or a displacement to be added (sign-extended) to the contents of the index register (XR).

- Notes:
1. The initial contents of the PC are lost.
 2. JMP indirect is a basic instruction.
 3. Address class 2.

7.9.3 Jump Through Pointer (JMPP) — Extended Instruction Set

The contents of the PC are set equal to the contents of the memory location addressed by the sum of the contents of memory location 100_{16} and the immediate value. The next instruction is fetched from the location designated by the new contents of the PC.

This instruction is faster than JMP when the new instruction is some distance from the jump instruction. It also is useful for accessing jump tables containing up to 16 words from one address.

Operation: $(PC) \leftarrow ((100_{16}) + \text{disp})$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JMPP	immed4

The operand field contains an immediate value that is added to the contents of memory location 100_{16} .

- Notes:
1. The initial contents of the PC are lost.
 2. JMPP is an extended instruction.
 3. Address class 7.

7.9.4 Jump to Subroutine Direct (JSR) — Basic Instruction Set

The contents of the PC are stored in the top of the stack. The effective address replaces the contents of the PC. The next instruction is fetched from the location designated by the new contents of the PC.

Operation: $(STK) \leftarrow (PC), (PC) \leftarrow EA$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JSR	address [(XR)]

The operand field contains a base page or PC-relative address or a displacement to be added (sign-extended) to the contents of the index register (XR).

- Notes:
1. A Return from Subroutine instruction or a Return from Interrupt instruction may be used to return to execution of the instruction directly following the subroutine jump in the main program.
 2. JSR is a basic instruction.
 3. Address class 2.

7.9.5 Jump to Subroutine Indirect (JSR) — Basic Instruction Set

The contents of the PC are stored in the top of the stack. The contents of the effective address replace the contents of the PC. The next instruction is fetched from the location designated by the new contents of the PC.

Operation: $(STK) \leftarrow (PC), (PC) \leftarrow (EA)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JSR	@address [(XR)]

The operand field contains a base page or PC-relative address.

- Notes:
1. A Return from Subroutine instruction or a Return from Interrupt instruction may be used to return to execution of the instruction directly following the subroutine jump in the main program.
 2. JSR is a basic instruction.
 3. Address class 2.

7.9.6 Jump to Subroutine Implied (JSRI) — Basic Instruction Set

The Jump to Subroutine Implied instruction enables a simplified subroutine jump to memory locations $FF80_{16}$ through $FFFF_{16}$. The contents of the PC are pushed onto the stack. The contents of the PC are then replaced by the address implied by the sum of the displacement and the number $FF80_{16}$.

Operation: $(STK) \leftarrow (PC), (PC) \leftarrow FF80_{16} + \text{ctl}$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JSRI	address

The operand field contains an address that is in the range $FF80_{16}$ through $FFFF_{16}$. The assembler will create the proper displacement for the machine instruction.

- Notes:
1. JSRI is a basic instruction.
 2. Address class 6.

7.9.7 Jump to Subroutine Through Pointer (JSRP) — Extended Instruction Set

The contents of the PC are stored in the top of the stack. The new contents of the PC are set equal to the contents of the memory location addressed by the sum of the contents of memory location 100_{16} and the immediate value. The next instruction is fetched from the location designated by the new contents of the PC.

Operation: $(STK) \leftarrow (PC), (PC) \leftarrow ((100_{16}) + \text{disp})$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JSRP	+immed

The operand field contains a positive immediate value that is added to the contents of memory location 100_{16} .

- Notes:
1. JSRP is an Extended Instruction.
 2. Address class 7.

7.9.8 Return from Subroutine (RTS) — Basic Instruction Set

The Return from Subroutine instruction is used primarily to return from subroutines entered by a jump to subroutine instruction. The contents of the PC are replaced by the sum of the immediate value and the contents of the top level of the stack. (The immediate value permits the user to skip around subroutine parameters, utilize different subroutine exits, etc.) Program control is transferred to the location specified by the new contents of the PC.

Operation: $(PC) \leftarrow (STK) + \text{immed}$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RTS	[+immed]

The operand field may contain an immediate value that is added to the contents of the top level of the stack.

- Notes: 1. The initial contents of the PC are lost.
2. RTS is a basic instruction.

7.9.9 Return from Interrupt (RTI) — Basic Instruction Set

The Return from Interrupt instruction is used primarily to exit from an interrupt routine. It behaves in exactly the same way as the RTS instruction except the RTI enables interrupts that are inhibited when an interrupt occurs. See appendix C.

The contents of the PC are replaced by the sum of the immediate value and the contents of the top level of the stack. Program control is transferred to the location specified by the new contents of the PC.

Operation: $(PC) \leftarrow (STK) + \text{immed}; \text{INTEN FLAG SET}$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RTI	+immed

The operand field may contain an immediate value that is added to the contents of the top level of the stack.

- Notes: 1. The Interrupt Enable flag (INTEN) is set.
2. The initial contents of the PC are lost.
3. RTI is a basic instruction.

7.9.10 Branch on Condition (BOC) — Basic Instruction Set

Several instructions set a condition code to indicate something about the result of the instruction's execution; for example, the carry and overflow flags are set after an add or subtract instruction. There are 16 possible condition codes (cc). The codes are listed in table 7-3 for both the IMP-16C and the IMP-16L.

At any time after the condition code has been set, it may be tested by using a Branch-on-Condition instruction. The four bits that identify the condition code are used as a jump condition multiplexer address to check for that condition. If the condition code is true, the value of the special address is added to the contents of the PC, and the sum is stored in the PC. Program control is transferred to the location specified by the new contents of the PC. If the condition is false, the next consecutive instruction is executed.

Operation; $(PC) \leftarrow (PC) + \text{disp}$ (sign extended from bit 7 through bit 15)

Coding Format: Operation Operand
 BOC immed4, spaddr

The operand field contains the number or a symbol that identifies the condition code and a special address with a value in the -127 to +128 range.

- Notes:
1. The initial contents of the PC are lost.
 2. PC is always incremented by 1 immediately following the fetching of an instruction, so the contents of PC during execution of an instruction is 1 greater than the address of that instruction. This must be considered during execution of the BOC instruction; for example, if the address of the BOC instruction is 100, then 101 is added to the special address.
 3. The special address field is a signed 8-bit number whose sign is extended from bit 7 through bit 15 to form a 16-bit number (including sign). Thus, the range of addressing with a BOC instruction is -127 to +128 relative to the address of the current instruction.
 4. BOC is a basic instruction.

Table 7-3. Branch On Condition Codes

IMP-16C	Condition Code (cc)	IMP-16L
Interrupt Line = 1	0	Interrupt Line = 1
(AC0) = 0	1	(AC0) = 0
(AC0) \geq 0	2	(AC0) \geq 0
Bit 0 of AC0 = 1	3	Bit 0 of AC0 = 1
Bit 1 of AC0 = 1	4	Bit 1 of AC0 = 1
(AC0) \neq 0	5	(AC0) \neq 0
CONTROL PANEL INTERRUPT LINE = 1	6	CONTROL PANEL INTERRUPT LINE = 1
CONTROL PANEL START = 1	7	CONTROL PANEL START = 1
STACK FULL LINE = 1	8	STACK FULL LINE = 1
INTERRUPT ENABLE = 1	9	INTERRUPT ENABLE = 1
CARRY/OVERFLOW = 1	10	CARRY/OVERFLOW = 1*
(AC0) \leq 0	11	(AC0) \leq 0
User	12	POA
User	13	SEL
User	14	User
User	15	User

*If the Select Flag (SEL) is set, overflow is tested. Otherwise, carry is tested.

7.10 SKIP INSTRUCTIONS (Basic and Extended Sets)

The seven skip instructions skip the next instruction if the specified condition is met. Skip instructions are commonly used for logical comparisons, table indexing, and test for zero. There are three types of skip instructions: memory, register, and bit and status flag. The skip instructions are:

- | | |
|-------------------------------------|-------------------------------------|
| 1. Memory References | 3. Bit and Status Flag |
| Increment and Skip if Zero ISZ | Skip if Status Flag True SKSTF |
| Decrement and Skip if Zero DSZ | Skip if Bit True SKBIT |
| 2. Register References | |
| Skip if Greater SKG | |
| Skip if Not Equal SKNE | |
| Skip if AND is Zero SKAZ | |

Caution should be taken when coding a skip instruction to prevent the skip condition from jumping into the displacement field of a double-word instruction (that is, the instruction following the skip instruction must be a single-word instruction).

7.10.1 Increment and Skip if Zero (ISZ) — Basic Instruction Set

The Increment and Skip if Zero instruction tests the contents of a word in memory. The contents of the effective address are incremented by 1. The new contents of the effective address are tested to determine if they equal zero. If the new contents of the effective address equal zero, the contents of the PC are incremented by 1, thus skipping the instruction designated by the initial contents of the PC.

Operation: $(EA) \leftarrow (EA) + 1$; if $(EA) = 0$, $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ISZ	address $[(XR)]$

The operand field specifies the direct effective address of the memory location to be tested.

Notes: 1. ISZ is a basic instruction.
2. Address class 1.

7.10.2 Decrement and Skip if Zero (DSZ) — Basic Instruction Set

The Decrement and Skip if Zero instruction tests the word in memory. The contents of the effective address are decremented by 1. The new contents of the effective address are tested to determine whether they equal zero. If the new contents of the effective address equal zero, the contents of the PC are incremented by 1, thus skipping the instruction designated by the initial contents of the PC.

Operation: $(EA) \leftarrow (EA) - 1$; if $(EA) = 0$, $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	DSZ	address $[(XR)]$

The operand field specifies the direct effective address of the memory location to be tested.

Notes: 1. DSZ is a basic instruction.
2. Address class 1.

7.10.3 Skip if Greater (SKG) — Basic Instruction Set

The Skip if Greater instruction compares the contents of a register and the contents of the effective address. The contents are compared on an algebraic basis with due regard to the signs of the two operands. If the contents of the register are greater than the contents of the effective address, the contents of the PC are incremented by 1, thus skipping the instruction designated by the initial contents of the PC.

Operation: If $(ACr) > (EA)$, $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SKG	register, address $[(XR)]$

The operand field contains the address of the working register and the direct effective address of the memory location to be compared.

Notes: 1. The initial contents of the PC are lost.
2. The contents of the register and the effective address are unaltered.
3. SKG is a basic instruction.
4. Address class 1.

7.10.4 Skip if Not Equal (SKNE) — Basic Instruction Set

The Skip if Not Equal instruction compares the contents of the working register and the contents of the effective address. If the contents are not equal, the contents of the PC are incremented, thus skipping the instruction designated by the initial contents of the PC.

Operation: If $ACr \neq (EA)$, $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SKNE	register, address $[(XR)]$

The operand field contains the address of the working register and the direct effective address of the memory location to be compared.

- Notes:
1. The initial contents of the PC are lost.
 2. The contents of the register and the effective address are unaltered.
 3. SKNE is a basic instruction.
 4. Address class 1.

7.10.5 Skip if AND Is Zero (SKAZ) — Basic Instruction Set

The Skip if AND Is Zero instruction causes the contents of the register and the effective address to be ANDed. If the result equals zero, the contents of the PC are incremented by 1, thus skipping the instruction designated by the initial contents of the PC.

Operation: If $[(ACr) \wedge (EA)] = 0$, $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SKAZ	accumulator, address $[(XR)]$

The operand field contains the address of working register 0 or 1, and the direct effective address of the memory location to be ANDed.

- Notes:
1. The initial contents of the PC are lost.
 2. The contents of the register and the effective address are unaltered.
 3. SKAZ is a basic instruction.
 4. Address class 1.

7.10.6 Skip if Status Flag True (SKSTF) — Extended Instruction Set

The Skip if Status Flag True instruction tests a status flag. If the specified status flag is true, the contents of the PC are incremented by 1, thus skipping the instruction designated by the initial contents of the PC.

Operation: If Status Flag $n = 1$, $(PC) \leftarrow (PC) + 1$; $SEL \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SKSTF	immed4

The operand field contains the numeric or symbol that identifies the status flag.

- Notes:
1. The contents of the status register are unaffected.
 2. The select flag is cleared.
 3. SKSTF is an extended instruction.

7.10.7 Skip if Bit True (SKBIT) — Extended Instruction Set

The Skip if Bit True instruction tests a bit in register AC0. If the designated bit is true, the contents of the PC are incremented by 1, thus skipping the instruction designated by the initial contents of the PC.

Operation: If $AC0_n = 1$, $(PC) \leftarrow (PC) + 1$; $SEL \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SKBIT	immed4

The operand field contains the number or symbol that identifies the status flag.

- Notes:
1. The contents of the register are unaffected.
 2. The select flag is cleared.
 3. SKBIT is an extended instruction.

7.11 SHIFT INSTRUCTIONS (Basic Set)

The shift instructions permit the programmer to shift data bits within a register. Four instructions comprise this group. All four instructions may be used with the Link (L) bit by setting the SEL flag. If the L bit is used, the SEL flag must be set with a Set Flag (SFLG) instruction before executing the shift or rotate instruction. All shift and rotate operations may be carried out with any of the four working registers, AC0, 1, 2, or 3. In shifting, sign bits are treated like any other bit.

7.11.1 Shift Left (SHL) — Basic Instruction Set

The Shift Left instruction is commonly used as a means of multiplying the contents of a register by a power of 2.

Shifting a word left 1 bit is equivalent to multiplying by 2. The SEL flag must be set so the carry (high-order bit) appears in the L bit. For example, the multiplication, $X'21 * X'53 = X'ABC$ may be effected by shifting $X'53$ left eight times, adding the result to a register each time the next bit of $X'21$ is 1, and tracking the L bit in another register.

The number in the working register is shifted left the number of bit positions specified by the immediate operand.

If the SEL flag is not set, the most significant bit is lost and a zero is shifted into the least significant bit for each shift.

If the SEL flag is set, the most significant bit is shifted into the L bit, the original content of L is lost, and a zero is shifted into the least significant bit for each shift.

Operation: $SEL = 0$ $(ACr_n) \leftarrow (ACr_{n-1})$, $(ACr_0) \leftarrow 0$

$SEL = 1$ $(L) \leftarrow (ACr_{15})$, $(ACr_n) \leftarrow (ACr_{n-1})$, $(ACr_0) \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SHL	register, immed

The operand field specifies the address of the register containing the word to be shifted and an immediate value specifying the number of shifts.

7.11.2 Shift Right (SHR) — Basic Instruction Set

The Shift Right instruction is commonly used as a means of dividing the contents of a register by 2 or to manipulate status indicators by testing consecutive bits and branching on bit 0 or bit 1.

The number in the working register is shifted right the number of bit positions specified by the immediate operand.

If the SEL flag is not set, the least significant bit is lost, and a zero replaces the most significant bit for each shift.

If the SEL flag is set, the least significant bit is lost and the contents of the L bit is shifted into the most significant bit. In this case, a zero is shifted into the L bit.

Operation: SEL = 0 $(ACr_{15}) \leftarrow 0, (ACr_n) \leftarrow (ACr_{n+1})$
 SEL = 1 $(L) \leftarrow 0, (ACr_{15}) \leftarrow (L), (ACr_n) \leftarrow (ACr_{n+1})$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	SHR	register, immed

The operand field specifies the address of the register containing the word to be shifted and an immediate value specifying the number of shifts.

7.11.3 Rotate Left (ROL) — Basic Instruction Set

The Rotate Left instruction is commonly used for multiprecision arithmetic and manipulating flags.

The contents of the register are shifted around to the left the number of bit positions specified by the immediate operand.

If the SEL flag is not set, the most significant bit is shifted into the least significant bit. In other words, bit 15 replaces bit 0, bit 0 replaces bit 1, and so on for each shift.

If the SEL flag is set, the most significant bit is shifted into the L bit and the former contents of the L bit is shifted into the least significant bit for each shift.

Operation: SEL = 0 $(ACr_0) \leftarrow (ACr_{15}), (ACr_n) \leftarrow (ACr_{n-1})$
 SEL = 1 $(ACr_0) \leftarrow (L), (L) \leftarrow (ACr_{15}), (ACr_n) \leftarrow (ACr_{n-1})$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ROL	register, immed

The operand field specifies the address of the register containing the word to be rotated and an immediate value specifying the number of rotations.

7.11.4 Rotate Right (ROR) — Basic Instruction Set

The Rotate Right instruction is commonly used for multiprecision arithmetic and manipulating flags.

The contents of the register are shifted around to the right the number of bit positions specified by the displacement.

If the SEL flag is not set, the least significant bit is shifted into the most significant bit for each rotation.

If the SEL flag is set, the least significant bit is shifted into the L bit and the former content of the L bit is shifted into the most significant bit for each rotation.

Operation: SEL = 0 $(ACr_{15}) \leftarrow (ACr_0)$, $(ACr_n) \leftarrow (ACr_{n+1})$
 SEL = 1 $(ACr_{15}) \leftarrow (L)$, $(L) \leftarrow (ACr_0)$, $(ACr_n) \leftarrow (ACr_{n+1})$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ROR	register, immed

The operand field specifies the address of the register containing the word to be rotated and an immediate value specifying the number of rotations.

7.12 INTERRUPT HANDLING INSTRUCTIONS (Extended Set)

The Interrupt Handling Instructions enable the user to control input in interrupt routines.

7.12.1 Interrupt Scan (ISCAN) — Extended Instruction Set

The Interrupt Scan instruction is used in servicing input/output devices that respond to the Interrupt Select Status order code described in the IMP-16L Users Manual. Before the Interrupt Scan instruction can be executed AC1 should be loaded with the Interrupt Select Status word, and AC2 should be loaded with the base address of a pointer table for the interrupt service routines. (The base address is 1 less than the address of the first pointer.)

If AC1 = 0, the select flag (SEL) is cleared, and the next instruction is executed. If AC1 \neq 0, then the select flag is cleared and AC1 is shifted right until a 1 is shifted out of bit zero. The number of shifts that occurred is added to the contents of AC2. The next memory location is skipped.

Operation: If (AC1) = 0, $SEL \leftarrow 0$;
 If (AC1) \neq 0, $SEL \leftarrow 0$;
 $AC1 \leftarrow [\text{shift AC1 right 1}]$ until 1 shifted out $(AC2) \leftarrow (AC2) +$
 number shifts; $(PC) \leftarrow (PC) + 1$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ISCAN	not used

Notes: 1. ISCAN is an extended instruction.

7.12.2 Jump to Level 0 Interrupt, Indirect (JINT) — Extended Instruction Set

The Jump to Level 0 Interrupt, Indirect instruction is used in servicing input/output devices on interrupt request level 0 as described in the IMP-16L Users Manual.

The contents of the PC are pushed onto the top of the stack. The new contents of the PC are set equal to the contents of the memory location whose effective address is formed by adding the special address field to 120_{16} .

Operation: $(STK) \leftarrow (PC); (PC) \leftarrow (120_{16} + \text{disp}); \text{INTEN} \leftarrow 0$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	JINT	spaddr

The operand field contains a special address whose value is added to 120_{16} to form an effective address.

- Notes:
1. The interrupt enable flag is cleared.
 2. JINT is an extended instruction.
 3. The special address operand must have a value between 0 and X'F.
 4. Address class 5.

7.13 INPUT/OUTPUT, HALT, AND CONTROL FLAG INSTRUCTIONS (Basic Set)

The five Input/Output, Halt, and Control Flag instructions are used to control input/output operations and control flags external to the RALUs. Appendix C discusses input/output programming in detail.

INPUT/OUTPUT INSTRUCTIONS

Input/output operations are carried out with the Register In and Register Out instructions. Functionally, these instructions are similar to the Load and Store instructions in that they address particular device locations and initiate data exchanges.

The effective address of an input/output device is determined by the sum of the contents of AC3 and the 7-bit control field of the RIN or ROUT instruction. Although a number of schemes are possible, the one described below as an example, has proved useful for most applications. The low-order three bits are used to define an input/output "order" and bits 3 to 6 are the device address. (Device addresses are installation specific and are not assigned by the programmer.) Each peripheral device decodes the address field of the input/output instruction, and, if the Read Peripheral or Write Peripheral flag is active, the appropriate device will respond.

The 4 bits of the device address field permit direct addressing of 16 devices; however, by loading AC3 with a 16-bit value before executing a RIN or ROUT instruction, many more devices may be accessed.

The three "order" bits permit eight possible auxiliary operations for each input/output class; for example, read data, read status, reset device, rewind tape, backspace, write data, and so forth. The assignment of the various orders is the responsibility of the systems programmer at the macro level. These auxiliary operations are fundamental to direct-memory-access (DMA) type operations.

7.13.1 Register In (RIN) — Basic Instruction Set

The Register In instruction is used to replace the contents of the address register with the sum of the immediate value and the contents of AC3. The new contents of the address register constitute the address of a peripheral device and a command, both of which are received by the peripheral device. The peripheral device responds by transferring the contents of its input/output register to AC0.

Operation: $(AR) \leftarrow \text{ctl} + (AC3), (AC0) \leftarrow (IOREG)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	RIN	+immed

The operand field contains an immediate value that is added to the contents of AC3 to form the address of a peripheral device.

7.13.2 Register Out (ROUT) — Basic Instruction Set

The Register Out instruction is used to replace the contents of the address register with the sum of the immediate value and the contents of AC3. The new contents of the address register constitute the address of a peripheral device and a command, both of which are received by the addressed peripheral device. The processor then transfer the contents of AC0 to the input/output register in the peripheral device.

Operation: $(AR) \leftarrow \text{ctl} + (AC3), (IOREG) \leftarrow (AC0)$

Coding Format:	<u>Operation</u>	<u>Operand</u>
	ROUT	+immed

The operand field contains an immediate value that is added to the contents of AC3 to form the address of a peripheral device.

7.13.3 Halt (HALT) — Basic Instruction Set

The Halt instruction causes the microprocessor to halt and remain halted until restarted.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	HALT	not used

CONTROL FLAG INSTRUCTIONS

The control flag instructions refer to control flags external to the RALUs. These flags should not be confused with RALU internal flags, which are referenced by PUSHF and PULLF.

Sixteen flag codes are provided. However, only control flags with addresses between 8 and 15 may be accessed with the control flag instructions. The flag address is 8 (binary 1000) greater than the corresponding flag code. This is done because control flags with flag addresses 0 through 7 are used for various input/output operations controlled by the microprocessor and are not useable by the programmer.

The control flag instructions set or pulse a control flag and transfer the control value (sign-extended from bit 7 through 15) to the address register. This word in the address register has no specified use and may be used as desired by the system programmer.

Flag codes are defined in table 7-4.

Table 7-4. Control Flags

FC	Flag Mnemonic	IMP-16L	IMP-16C
0	F8	User Specified	User Specified
1	INT EN	Interrupt Enable	Interrupt Enable
2	SEL	Select (SEL)	Select (SEL)
3	F11	User Specified	User Specified
4	F12	User Specified	User Specified
5	F13	SCPIE	User Specified
6	F14	User Specified	User Specified
7	F15	User Specified	User Specified

Flag code 1 is assigned to the Interrupt Enable flag. Interrupts are enabled when the flag is set.

Flag code 2 is assigned to the Select (SEL) flag that controls whether or not the Link (L) status bit is activated during multiply, divide, shift and/or rotate execution.

Flag code 5 is assigned to the Special Control Panel Interrupt Enable (SCPIE - see IMP-16L Users Manual).

The remaining flag codes are available to the user. Pins may be wired for these flags as described in the appropriate IMP-16 users manual. Typically, these flags are used to transmit control signals to or receive control signals from external devices.

7.13.4 Set Flag (SFLG) — Basic Instruction Set

The Set Flag instruction sets the control flag designated by the flag code and replaces the contents of the address register with the value of the immediate operand. Bit 7, (containing a 0) is extended through bit 15.

Operation: FC set, (AR) \leftarrow ctl (bit 7 extended through bit 15; that is, bits 8 through 15 set to 0)

Coding Format: Operation Operand
 SFLG immed3, [+immed]

The operand field specifies the flag code which may be followed by an immediate operand that replaces the contents of the address register.

7.13.5 Pulse Flag (PFLG) — Basic Instruction Set

The Pulse Flag instruction pulses the control flag designated by the flag code and replaces the contents of the address register with the value of the immediate operand. Bit 7, (containing a 1) is extended through bit 15. Pulsing a control flag sets the flag at T2 and resets it at T6 during the same microcycle. The flag remains reset until again set or pulsed.

Operation: FC pulsed, (AR) \leftarrow ctl (bit 7 extended through bit 15; that is, bits 8 through 15 set to 1)

Coding Format: Operation Operand
 PFLG immed3, [+immed]

The operand field specifies the flag code which may be followed with an immediate operand that replaces the contents of the address register.

Chapter 8

ASSIGNMENT STATEMENT

The assignment statement assigns a value to a symbol. It replaces a number of directives (EQU, ORG, SET, and so forth) found in other assemblers. The assignment statement may be preceded by a series of labels and followed by a comment.

Coding Format:	<u>Label</u>	<u>Operation</u>	<u>Operand</u>
	label:	Symbol =	expression

The effect of an assignment statement is to take the value of the expression and its mode on the right side of the equal sign and give it to the symbol or symbols on the left. For example: RETURN = OD ;SYMBOL HAS CARRIAGE RETURN CODE.

The assignment statement provides the ability to set the location counter or to refer to the current value of the location counter in an expression. The special symbol, ". " is used to specify the location counter. A location counter reference in an assignment statement designates the location counter value for the sector in which the reference appears. The location counter symbol may appear on either side of the equal sign. If it appears on the left, it is assigned the value on the right side of the equal sign. The programmer may refer to the current setting of the location counter by inserting the ". " immediately after the equal sign. The period represents the location of the first word of currently available storage. Assignment statements using the location counter symbol are coded as free-form statements. For example:

```

      . =20          ;SET LOCATION COUNTER TO 20
                        (Note: location counter must be in absolute mode)
TABLE: . =. +10      ;RESERVE 10 LOCATIONS FOR TABLE

```

When the ". " appears on the left of the equal sign, the mode of the expression on the right must be the same as the current mode of the location counter.

NOTE

The only way to change the mode of the location counter is through "section directive statements."

In addition, if the ". " appears on the left or if the symbol on the left is a global symbol, the expression on the right must be defined during the first pass so label assignments may be made.

If the symbol on the left is not ". " or a global symbol, then the expression to the right need not have a value during the first pass, but it must have a value during the second pass. This permits only one level of forward referencing. An example of more than one level of forward referencing follows:

```

A = B+2 ; } expressions undefined during pass 1
          }
B = C-1 ; } expression undefined during pass 2
          }
C = 25

```

Chapter 9

DIRECTIVE STATEMENTS

The directive statements control the assembly process and may generate data in the object program.

The directive operator may be preceded by one label or more, and may be followed by a comment. It occupies the operator field and is followed by no operand, one operand, or more operands depending on the particular operator.

Assembler directive operators and their functions are summarized in table 9-1. Note that all directive operators begin with a period for easy visual differentiation from the instruction operator mnemonics in the output listing. Each directive operator is described in more detail on the following pages.

Table 9-1. Summary of Assembler Directives

Directive Name	Function
.TITLE	Names a load module
.ASECT	Specifies the start of an absolute section
.BSECT	Specifies the start of a base sector relocatable section
.TSECT	Specifies the start of a top sector relocatable section
.END	Physical end of source program
.LIST	Listing output control
.SPACE	Space 'n' lines in output listing
.PAGE	Output listing to top-of-form
.WORD	16-bit word data generation
.ASCII	Data generation for character strings
.GLOBL	Identifies global symbols
.LOCAL	Establishes a new local symbol region
.IF	Conditional assembly directives
.ELSE	Conditional assembly directives
.ENDIF	Conditional assembly directives
.FORM	Field specification
.EXTD	Allows extended instruction set to be used in an assembly

9.1 TITLE DIRECTIVE (.TITLE)

The .TITLE directive identifies the relocatable load module in which it appears with a symbolic name and an optional definitive title. If a .TITLE directive does not appear in the program, the load module is given the name, "MAINPR." If more than one .TITLE directive is coded, the last one encountered is the directive used.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.TITLE	symbol [,string]

The symbolic name and the string must meet the symbol and string construction restrictions (chapter 4).

9.2 PROGRAM SECTION DIRECTIVES (.ASECT, .BSECT, .TSECT)

.ASECT — Absolute Sector
 .BSECT — Base Sector
 .TSECT — Top Sector

The three program section directives enable the user to create a program in sections, producing a load module that is absolute, base-sector-relative, top-sector-relative, or a combination of the three.

Execution of a program section directive during the assembly process causes the location counter to be set to the new mode and to assume a value that was its last value in that mode. Initially, the location counter of the assembler is in the T mode (top-sector-relocatable), and the current value for each mode is zero. A program section directive is in effect until a program section directive to the contrary is encountered.

All labels and the symbol " . " for the location counter assume the mode of the current program section directive.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.ASECT	not used
	<u>Operation</u>	<u>Operand</u>
	.BSECT	not used
	<u>Operation</u>	<u>Operand</u>
	.TSECT	not used

9.3 END DIRECTIVE (.END)

The .END directive signifies the physical end of the source program. The optional address in the operand field may be used to indicate an execution address to the loader. In other words, it causes a branch to the address of the first instruction to be executed (entry point as opposed to load point) after the load is complete.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.END	[address]

9.4 LIST DIRECTIVE (.LIST)

The .LIST directive is used to suppress and reinstate the output listing. Initially, the assembler is in the list mode. When a .LIST directive is encountered, the assembler suppresses the output listing if the value of the immediate operand in the operand field is less than or equal to zero. Otherwise it reinstates the output listing.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.LIST	immed

9.5 SPACE DIRECTIVE (.SPACE)

The .SPACE directive causes a skip forward the specified number of lines on the output listing.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.SPACE	immed

The value of the immediate operand specifies the number of lines to be spaced forward.

9.6 PAGE DIRECTIVE (.PAGE)

The .PAGE directive is used to space forward to the top of the next page. The optional string is used as the current page title and is printed on each page until another Page Directive is encountered with a new string to replace the old title. No action takes place (except for a new page title) if the .PAGE directive is issued immediately after an assembler generates top-of-page request.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.PAGE	[string]

9.7 WORD DIRECTIVE (.WORD)

The .WORD directive generates 16-bit data words in successive memory locations. Each expression in a .WORD directive is evaluated, and its value is placed in the next available memory location.

If a label appears in the label field of a .WORD directive statement, it refers to the address of the memory location of the first value.

In all expressions appearing in a single .WORD directive statement, the special symbol " . " will have the value of the location counter that corresponds to the initial expression.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.WORD	expression [, expression, ... expression]

9.8 ASCII DIRECTIVE (.ASCII)

The .ASCII directive generates data in successive memory locations by translating the characters in the string(s) into their 7-bit ANSI equivalent code (see appendix A). The characters in the string must be enclosed in single quote marks (').

The first character in each string is placed in the high-order byte of the next available memory location. The second character is placed in the low-order byte. The third character is placed in the high-order byte of the next word, and so on. If there is an odd number of characters in a string, the last low-order byte is filled with a blank code (20₁₆).

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.ASCII	string [,string,.....string]

9.9 GLOBAL DIRECTIVE (.GLOBL)

The .GLOBL directive lists a set of symbols as being global to all load modules that are linked and loaded together. This is the mechanism by which individually assembled programs can communicate with one another.

Each symbol in the operand field is marked by the assembler as a global symbol. If the symbol is within the current assembly it may be referred to by other load modules. If the symbol is not within the current assembly, it is assumed to be defined in another assembly, and references to this symbol will be established at load time.

Any number of .GLOBL directive statements may occur within a single assembly. They are treated as a single .GLOBL directive statement with a large number of symbol operands.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.GLOBL	symbol [,symbol,symbol...symbol]

9.10 LOCAL DIRECTIVE (.LOCAL)

The .LOCAL directive establishes a new program region for local symbols (symbols beginning with a dollar sign). Designated symbols between two .LOCAL directive statements have the meaning assigned to them only within that particular region of the program. Note: A local .LOCAL directive is assumed at the beginning and end of a program, thus, one .LOCAL directive in the module splits the program into two regions.

If the first character of a symbol is a dollar sign (\$), the assembler attaches a unique character to the end of the symbol. Initially, this character is the exclamation point (!). The value of the added character is advanced by one with the letter "Z" being the last legal value. Therefore, up to 58 local symbols can appear in one section (See appendix A).

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.LOCAL	not used

9.11 CONDITIONAL ASSEMBLY DIRECTIVES (.IF, .ELSE, .ENDIF)

The conditional assembly directives selectively assemble portions of a source program based on the value of the expression in the .IF directive statement.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.IF	expression
	<u>Operation</u>	<u>Operand</u>
	.ELSE	not used
	<u>Operation</u>	<u>Operand</u>
	.ENDIF	not used

All source statements between an .IF directive and its associated .ENDIF directive are defined as an .IF-.ENDIF block. These blocks can be nested to a depth of ten. The .ELSE directive can be optionally included in an .IF-.ENDIF block. It segments the block into two parts. The first part of the source statements is assembled if the .IF expression is greater than zero; otherwise, the second part is assembled. When an .ELSE directive is not included in a block, the block will be assembled only if the expression is greater than zero. The example in figure 9-1 illustrates how the conditional assembly directives operate.

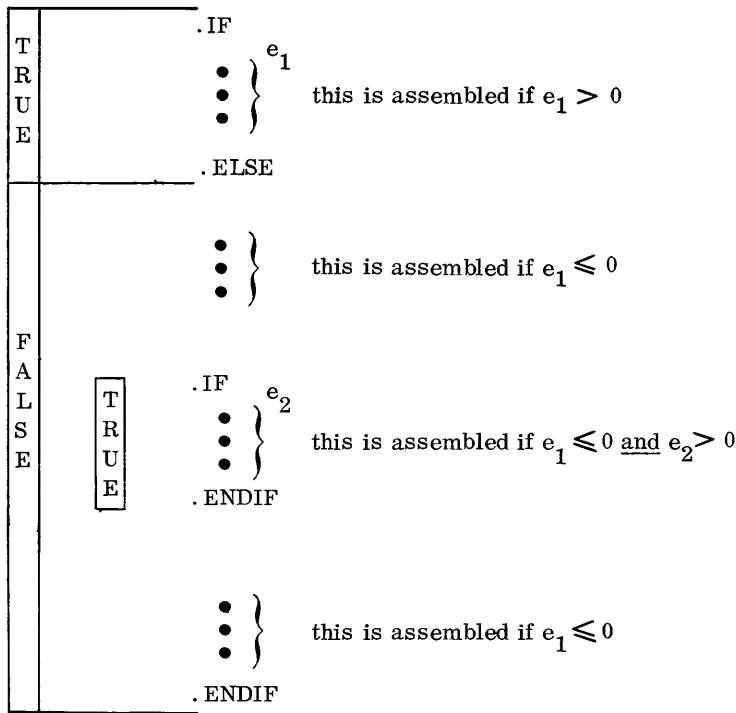


Figure 9-1. Example of Conditional Assembly Directives

9.12 FORM DIRECTIVE (. FORM)

The . FORM directive specifies a field format for a 16-bit word and optionally presets bits of the word to an initial value. The . FORM directive may be used for such purposes as generating special instructions not recognized by the assembler.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	. FORM	symbol, exp [(exp)] , [(exp [(exp)])]

The symbol operand is the name of the word format and can be used to invoke the format by placing it in the operation field of a statement. Note: symbols assigned values by a . FORM directive cannot be used in expressions or as global symbols.

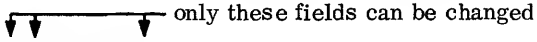
The expression(s) following the symbol specify field lengths within the word and optionally, values for the specified field lengths starting at the most significant bit. If a value is specified for a field it must be enclosed in parentheses and immediately follow the length attribute. For example,

```
. FORM      CK, 4, 8, 4(X'A)
```

divides the word into three fields: one 4-bit field, one 8-bit field, and one 4-bit field with the preset value, X'A. Note: fields with preset values cannot be changed when the format is evoked.

The following examples illustrate the use of the . FORM directive.

1. To generate a BCD constant



```
. FORM BCD, 4, 4, 4(X'F), 4
```

When the BCD form is invoked by

```
BCD X'C, X'A, X'E
```

it causes the 16-bit CAFE₁₆ to be generated.

2. To generate I/O instructions:

```
TTY = 1
READ = 2
WRITE = 7
.
.
.
. FORM RDCHAR, 8(04), 1(0), 4(TTY), 3(READ)
. FORM WRCHAR, 8(06), 1(0), 4(TTY), 3(WRITE)
.
.
.
RDCHAR
.
.
.
WRCHAR
```

The instructions generated would be X'040A (RIN) and X'060F (ROUT); these instructions would read a character from TTY (with a parallel interface) and write a character to TTY.

3. To generate conditional branch instructions:

```

ZRO = 1
NZRO = 5
.FORM  BZ, 4(1), 4(ZRO), 8
.FORM  BNZ, 4(1), 4(NZRO), 8
.
.
.
BZ      LABEL-. -1
.
.
.
BNZ     LAB2-. -1

```

The object code generated would be:

```

      BZ:  BOC   REQO, LABEL
      BNZ: BOC   NREQO, LAB2

```

9.13 EXTENDED INSTRUCTION DIRECTIVE (.EXTD)

In order to avoid inadvertent use of the extended instruction set by a user who does not have a second CROM installed, the assembler will normally flag any occurrence of an extended instruction with a diagnostic message. The instruction will, however, be assembled correctly if it is structured correctly.

Execution of the .EXTD instruction directive causes the printing of the error message to be suppressed.

Coding Format:	<u>Operation</u>	<u>Operand</u>
	.EXTD	not used

Chapter 10

IMP-16 RESIDENT ASSEMBLER OPERATING PROCEDURE

The IMP-16 resident assembler is a 3-pass assembler that may be run on an IMP-16L or IMP-16P microcomputer and must read the same program for each pass, as follows.

Pass 1: During pass 1 the assembler assigns locations to the labels.

Pass 2: Pass 2 is optional and exists if and only if a listing is requested (which is the normal mode).

Pass 3: Pass 3 is an optional pass and exists if and only if an object module is to be punched on paper tape. (OM option specified on control statement)

10.1 ASSEMBLER LOADING PROCEDURE

The assembler is loaded in the normal manner by the absolute card reader loader (ABSCR) or the paper tape absolute loader (see IMP-16 Utilities Reference Manual). Upon completion of loading the assembler will type out

NSC IMP-16 ASSEMBLER
MEMORY =

to which the user must respond with his memory configuration (in units of K words, where 1K = 1024 words). The possible responses are as follows

a:b, c:d	RET	memory configured in two disjointed regions
a:b	RET	memory configured in one continuous region
	RET	default memory configuration 0 to 4K

where a, b, c, and d are integers 0 to 64 that indicate the memory configuration to be

aK to bK-1
and
cK to dK-1

For example,

0:4, 60:64 **RET**

will indicate a memory configuration of 0 to 4095 and 61440 (60K) to 65535 (64K-1).

The assembler uses this memory configuration information so that all memory that is not occupied by the assembler itself will be used for its symbol table. These numbers may vary depending upon the latest assembler release and the length of the user's symbols. Three words of symbol table are required for each symbol which contains four or less characters. Four words of symbol table are required for symbols containing more than four characters.

10.2 ASSEMBLING A PROGRAM

At the beginning of each assembly, the assembler will initialize to all of the default modes and will prompt with ".ASM" to which the user must respond with his control options. The control options are defined below and are separated by commas and terminated by a carriage return. If an erroneous control message is typed in, then the assembler will reinitialize and reprompt.

KB	Indicates keyboard input (this is the default mode).
PT	Indicates paper tape input.
CR	Indicates card reader input.
OM	Indicates object module request. The default mode is no object module and, therefore, there is no pass 3.
EL	Indicates error listing. During pass 2, only the lines with errors will be printed.
NL	Indicates no listing. The default mode assumes a listing. If there is no listing, pass 2 will be eliminated.
X	Indicates extended instructions are legal. The default mode is extended instructions illegal.
NC	Indicates no comments to be printed which will speed up the listing pass. The default mode is for comments to be printed.

The default mode (keyboard input, full listing, no extended instructions, and no object module) is always initialized at the beginning of each new assembly.

At the end of each pass, the source program must be reloaded for the next pass, and the assembly automatically begins the subsequent pass. At the beginning of the third pass, the assembly halts for the operator to turn the paper tape punch on. After the operator turns the paper tape on, he presses the run button to continue. At the completion of pass 3, the assembler again halts for the operator to turn the paper tape punch off and to press the run button to complete the assembly process. After assembling a program, the assembler reinitializes and prompts with ".ASM" for the next assembly.

The resident assembler also permits the user to specify the above assembly controls in his source program. This is done by using the .ASM directive with the above control options. If the .ASM directive appears in the source program, it is usually the first record. However, it may appear anywhere in his program: for example, to change the source device. Other than to change the source device, it is not recommended that the .ASM directive appear anywhere in the program except at the first record. The following is an example of a .ASM directive:

```
.ASM      X,OM,NC
```

10.3 CARD READER INPUT

A source program from the card reader contains one statement per card. Columns 1 to 72 contain the statement and columns 73-80 may contain identification information which is ignored by the assembler. A carriage return character causes immediate termination of the source statement. Otherwise, the source statement is terminated after column 72.

10.4 PAPER TAPE INPUT

A source program from paper tape contains one statement per record. Carriage return characters must terminate each record, and the record may not contain more than 72 characters.

10.5 KEYBOARD INPUT

A source program entered from the keyboard should be formatted one statement per record. Carriage return characters must terminate each record and the record may not contain more than 72 characters.

The assembler prompts for each statement from the keyboard with a statement number followed by an asterisk (*).

10.6 KEYBOARD/PAPER TAPE SPECIAL EDITING CHARACTERS

Assembler input from the keyboard or paper tape allows the following editing characters:

NULL	(00)	Ignored
RUBOUT	(FF)	Ignored
LJNE FEED	(0A)	Ignored
←	(5F)	Delete previous character (backspace)
ALT	(7D)	Delete source line
CR	(0D)	Terminates each source line

The above editing characters are processed as such, even if they appear with a character string.

10.7 OBJECT LISTING

Figure 10-1 illustrates typical object listings from the IMP-16 resident assembler. Only the first 53 columns of each source statement are listed, although the entire source statement is processed.

If the "no comment" (NC) mode of listing output has been specified, one of the two following conditions will hold:

1. If the comment begins in column 1 of the card, the card is completely ignored by the resident assembler and no actions are taken.
2. If the comment begins in other than column 1, the line is numbered and listed up to, but not including, the comment field.

10.8 RELOCATABLE LOAD MODULE (RLM)

Each object module record is punched on paper tape in the following format:

```

8 null frames
Start of Text Character (02)
Object Module Record (see Chapter 6 (6.3.3), Relocatable Load Module
(Output))
Carriage Return (0D)
Line Feed (0A)
```

The first record is preceded by 8 additional null frames and the last record is followed by 64 null frames.

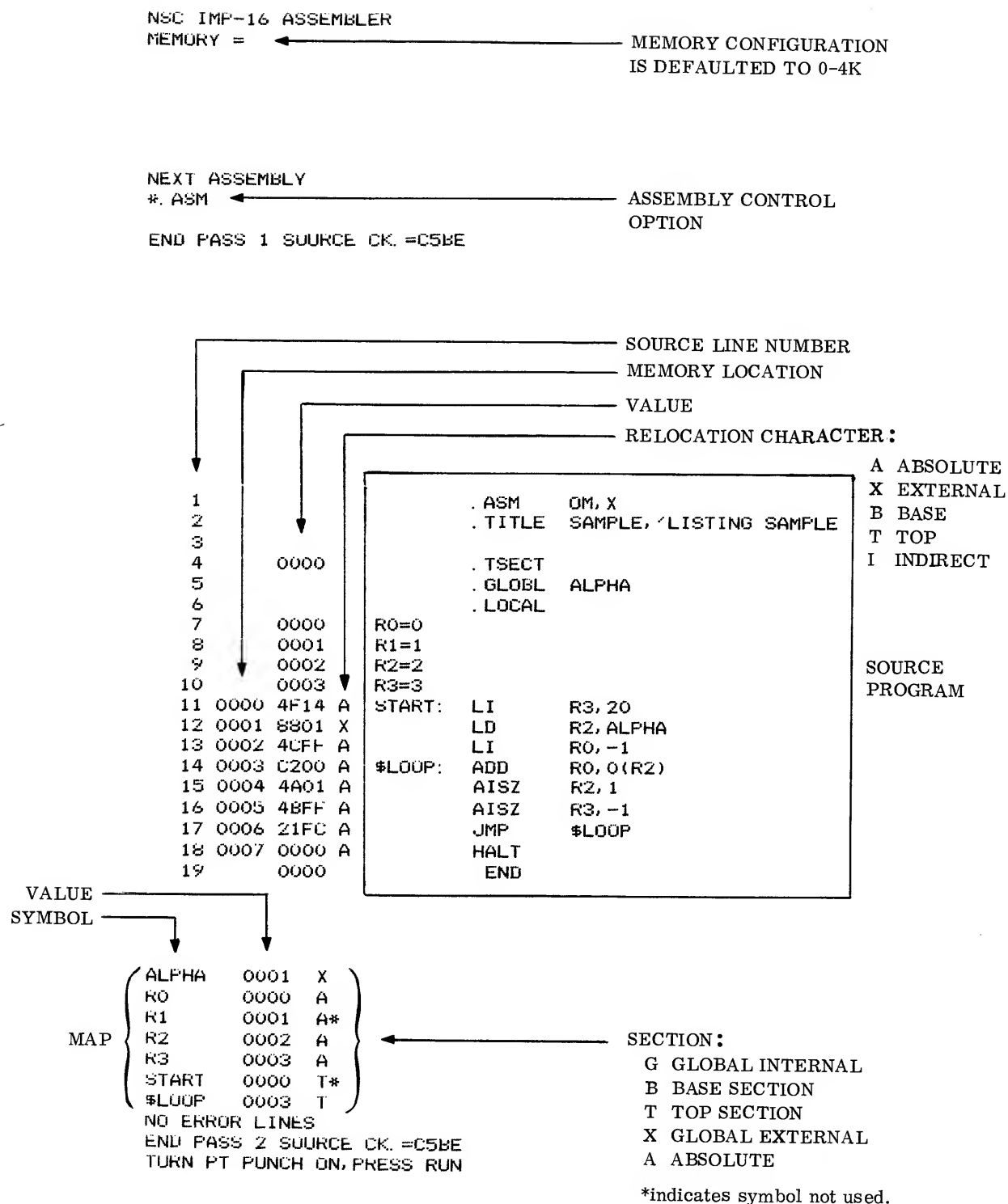


Figure 10-1. Sample Listing of Resident Assembler

Appendix A

CHARACTER SETS

Table A-1 contains the 7-bit hexadecimal code for each character in the ANSI character set. The printable characters in this set may be set-up as program data by use of the .ASCII directive (see chapter 9). The remaining characters may be set up in hexadecimal constants with a .WORD directive (see chapter 9). Table A-2 contains the legend for nonprintable characters.

Table A-1. ANSI Character Set in Hexadecimal Representation

Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number
NUL	00	!	21	A	41
SOH	01	"	22	B	42
STX	02	#	23	C	43
ETX	03	\$	24	D	44
EOT	04	%	25	E	45
ENQ	05	&	26	F	46
ACK	06	'	27	G	47
BEL	07	(28	H	48
BS	08)	29	I	49
HT	09	*	2A	J	4A
LF	0A	+	2B	K	4B
VT	0B	,	2C	L	4C
FF	0C	-	2D	M	4D
CR	0D	.	2E	N	4E
SO	0E	/	2F	O	4F
SI	0F	0	30	P	50
DLE	10	1	31	Q	51
DC1	11	2	32	R	52
DC2	12	3	33	S	53
DC3	13	4	34	T	54
DC4	14	5	35	U	55
NAK	15	6	36	V	56
SYN	16	7	37	W	57
ETB	17	8	38	X	58
CAN	18	9	39	Y	59
EM	19	:	3A	Z	5A
SUB	1A	;	3B	[5B
ESC	1B	<	3C	\	5C
FS	1C	=	3D]	5D
GS	1D	>	3E	↑	5E
RS	1E	?	3F	←	5F
US	1F	@	40	`	60
SP	20				

Table A-1. ANSI Character Set in Hexadecimal Representation (Cont.)

Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number	Character	7-Bit Hexadecimal Number
a	61	k	6B	u	75
b	62	l	6C	v	76
c	63	m	6D	w	77
d	64	n	6E	x	78
e	65	o	6F	y	79
f	66	p	70	z	7A
g	67	q	71		7B
h	68	r	72		7C
i	69	s	73	ALT	7D
j	6A	t	74	ESC	7E
				DEL, RUBOUT	7F

Table A-2. Legend for Nonprintable Characters

Character	Definition	Character	Definition
NUL	NULL	SO	SHIFT OUT
SOH	START OF HEADING; ALSO	SI	SHIFT IN
	START OF MESSAGE	DLE	DATA LINK ESCAPE
STX	START OF TEXT; ALSO EOA,	DC1	DEVICE CONTROL 1
	END OF ADDRESS	DC2	DEVICE CONTROL 2
ETX	END OF TEXT; ALSO EOM,	DC3	DEVICE CONTROL 3
	END OF MESSAGE	DC4	DEVICE CONTROL 4
EOT	END OF TRANSMISSION (END)	NAK	NEGATIVE ACKNOWLEDGE
ENQ	ENQUIRY (ENQRY); ALSO WRU	SYN	SYNCHRONOUS IDLE (SYNC)
ACK	ACKNOWLEDGE. ALSO RU	ETB	END OF TRANSMISSION
BEL	RINGS THE BELL		BLOCK
BS	BACKSPACE	CAN	CANCEL (CANCL)
HT	HORIZONTAL TAB	EM	END OF MEDIUM
LF	LINE FEED OR LINE SPACE	SUB	SUBSTITUTE
	(NEW LINE): ADVANCES	ESC	ESCAPE. PREFIX
	PAPER TO NEXT LINE	FS	FILE SEPARATOR
	BEGINNING OF LINE	GS	GROUP SEPARATOR
VT	VERTICAL TAB (VTAB)	RS	RECORD SEPARATOR
FF	FORM FEED TO TOP OF	US	UNIT SEPARATOR
	NEXT PAGE (PAGE)	SP	SPACE
CR	CARRIAGE RETURN TO		

Appendix B

STATUS BITS IN ARITHMETIC OPERATIONS

This section describes how status bits are used in arithmetic operations. Key programming aspects of status bit use are explained, but complete mathematical algorithms are not provided.

Arithmetic operations may be signed or unsigned. Consider first one word. Unsigned, its numbering range is:

$$\begin{array}{ll} \text{from } 0_{10} \text{ (X'0000)} & = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \text{to } 65535_{10} \text{ (X'FFFF)} & = 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \end{array}$$

Signed, the high-order bit is 0 for + (plus), 1 for - (minus), and the numbering range is:

$$\begin{array}{ll} +32767_{10} & = 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 0_{10} & = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ -1_{10} & = 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ -32768_{10} & = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

B.1 ARITHMETIC WITH UNSIGNED DATA WORDS

Unsigned arithmetic uses the carry bit, but not the overflow bit. For example:

$$\begin{array}{llll} + \text{X'2E00} & = & 0\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ \text{X'5200} & = & \underline{0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0} & \\ = \text{X'8000} & & 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \text{CY} = 0, \text{ no carry.} \\ \\ + \text{X'AE00} & = & 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ \text{X'5200} & = & \underline{0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0} & \\ = \text{X'10000} & & 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \text{CY} = 1 \end{array}$$

Consider subtraction using twos-complement arithmetic:

$$\text{X'AE00} - \text{X'5200} = \text{X'AE00} + \text{X'ADFF} + \text{X'0001} = \text{X'5000}$$

$$\begin{array}{llll} + \text{X'AE00} & = & 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ + \text{X'ADFF} & = & 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & \\ \text{X'0001} & = & \underline{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1} & \\ = \text{X'5C00} & & 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \text{CY} = 1 \text{ answer positive} \end{array}$$

$$\text{X'5200} - \text{X'AE00} = \text{X'5200} + \text{X'51FF} + \text{X'0001} = \text{X'A400} = - \text{X'5C00}$$

$$\begin{array}{llll} \text{X'5200} & = & 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ \text{X'5100} & = & 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & \\ \text{X'0001} & = & \underline{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1} & \\ \text{X'A400} & = & 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \text{CY} = 0 \text{ answer negative} \\ \text{Ones complement} & = & 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & \text{so take twos complement} \\ \text{Twos complement} & = & 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ & = & \text{X'5C00} & \end{array}$$

Rules for addition and subtraction using unsigned data words are as follows:

1. Ignore the OV bit.
2. In addition, if CY is set, add 1 to the next high order digit.
3. In subtraction, if CY is set, the answer is positive. If CY is reset, the answer is negative and is present in its twos-complement form.

In multiword arithmetic the above three rules apply to the leftmost (terminal or high-order) word. Between lower order words, the CY bit is always treated as a carry into the low-order bit of the next word:

$$\begin{array}{rcl}
 + \text{X}'1354\text{E}705 & = & 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \\
 \text{X}'24\text{C}33589 & = & \underline{0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1} \\
 & & 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0
 \end{array}$$

CY = 1

Carry into next word

B.2. MULTIPLICATION AND DIVISION

Consider multiplication and division of unsigned multibyte numbers. Two techniques are possible:

1. Repetitive addition/subtraction
2. Use of the shift instruction

Repetitive addition provides the simplest but slowest form of multiplication; for example, $(X'2E) * (X'74)$ may be generated by adding $(X'74)$ $X'2E$ times. Likewise, repetitive subtraction provides the simplest but slowest form of division. For example, to divide $(X'82)$ by $(X'21)$, subtract $(X'21)$ from $(X'82)$ repetitively until the answer is negative and count the number of subtractions that leave a positive answer.

Using shift instructions provides faster multiplication and division. Consider $(X'21) * (X'53)$.

$$X'21 * X'5300 = X'AB300$$

Recall that shifting a word left 1 bit is equivalent to multiplying by 2. A carry will appear in the L bit. The above multiplication may be effected by shifting $X'5300$ left eight times, adding the result to an accumulator each time the next bit of $X'21$ is 1, and tracking the L bit in another accumulator:

EXAMPLE OF MULTIPLICATION ROUTINE

	AC0	AC1	AC2	AC3
	$X'21$	$X'5300$	Link Accumulator	Answer Accumulator
Start:	---- 00100001	0101001100000000	0000000000000000	0000000000000000
Step 1	-----	0101001100000000	0000000000000000	0101001100000000
Step 2	-----	1010011000000000	0000000000000000	0101001100000000
Step 3	-----	0100110000000000	0000000000000001	0101001100000000
Step 4	-----	1001100000000000	0000000000000010	0101001100000000
Step 5	-----	0011000000000000	0000000000000101	0101001100000000
Step 6	-----	0110000000000000	0000000000001010	1011001100000000

- Step 1: Test AC0 0-bit; it is 1, so add AC1 to AC3.
 Step 2: Shift AC1 and AC2 left one bit. L bit is 0, no action. Test AC0 1-bit; it is 0, so no action.
 Step 3: Shift AC1 and AC2 left one bit. L bit is 1, so increment AC2. Test AC0 2-bit; it is 0, so no action.
 Step 4: Shift AC1 and AC2 left one bit. L bit is 0, no action. Test AC0 3-bit; it is 0, so no action.
 Step 5: Shift AC1 and AC2 left one bit. L bit is 1, so increment AC2. Test AC0 4-bit; it is 0, so no action.
 Step 6: Shift AC1 and AC2 left one bit. L bit is 0, no action. Test AC0 5-bit; it is 1, so add AC1 to AC3. If Carry, add 1 to high-order answer. Add Link Accumulator to high-order answer.

All remaining bits of AC1 are 0, so terminate AC2 and AC3 provide the answer, $X'0AB3$.

Since the multiplication routine described above uses a number of important programming techniques, a sample program is given below, with program comments.

A divide routine would follow virtually the same logic, but would use a left shift rather than a right shift on AC1.

EXAMPLE OF MULTIPLICATION ROUTINE

```

; one word * one word multiplication with 2-word answer.
; clear AC2 to hold link accumulation (becomes high-order portion of answer)

P10:      RXOR      AC2, AC2
          RXOR      AC3, AC3      ; Clear AC3 to hold L O answer
          ST        AC3, EXTND    ; Clear high-order answer

; Load multiplier into scratch word.
; Load AC1 with multiplicand. To make the multiplication routines general
; purpose, both numbers are loaded indirect, via addresses stored in Page 0.
; Therefore before executing this multiplication routine, the addresses of the
; multiplier and multiplicand words must be loaded at MUL1 and MUL2 on page 0.

          LD        AC1, @MUL1    ; Load multiplicand
          LD        AC0, @MUL2    ; Load multiplier
          ST        AC0, SCR      ; Save AC0 contents in scratch word
          LI        AC0, 1        ; Initialize the bit counter by setting
          ST        AC0, BCNT     ; the LO bit to 1 and all other bits to 0.
          SKAZ      AC0, SCR
          JMP       . +2          ; Bit set
          JMP       $M20         ; Bit not set

; Addition step for 1 bit multiplier begins here.

$M10:     RADD      AC1, AC3      ; Add AC1 to AC3.
          PFLG      SEL          ; Test Carry Bit
          BOC       CYOV, . +2
          JMP       $M15
          ISZ       EXTND        ; Add Carry to high-order answer
$M15:     LD        AC0, EXTND    ; Form high-order portion of answer.
          RADD      AC2, AC0
          ST        AC0, EXTND
$M20:     SFLG      SEL          ; Set SEL to activate LNK on shift.
          SHL       AC2, 1        ; Shift AC2 left one bit.
          SHL       AC1, 1        ; Shift AC1 left one bit.
          PUSHF     ; Shift status to AC0
          PULL      AC0          ; Test link bit
          BOC       PZRO, $M30
          AISZ      AC2, 1        ; Add bit in Link to Link Accumulator
$M30:     LD        AC0, BCNT     ; Restore bit count
          SHL       AC0, 1        ; Shift bit count left
          ST        AC0, BCNT
          SKAZ      AC0, SCR      ; Bit set in Multiplier
          JMP       $M10         ; Yes. Perform Next Step.
          AISZ      AC0, 0        ; No. Any more bits to test?
          JMP       $M20         ; Yes. Return for next shift.
          LD        AC2, EXTND    ; No. Fetch high-order portion of answer.
          RTS       ; Terminate multiplication.
SCR:      . =. +1              ; Multiplier stored here.
BCNT1:    . =. +1              ; Bit counter.
EXTND:    . WORD      0        ; High-order answer.

```

Appendix C

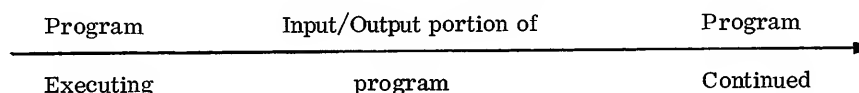
INPUT/OUTPUT PROGRAMMING

The programming of data transfers between read/write memory and peripheral devices is generally classified as input/output programming. Depending on how significant input/output operations are in the overall program, different approaches to input/output program implementation are recommended, as described in the following sections. First, it is necessary to clearly understand the differences between programmed input/output and interrupt input/output, as explained in C.1; next, section 2 describes generally accepted programming techniques that make input/output programming fast and efficient.

C.1 PROGRAMMED INPUT/OUTPUT AND INTERRUPT INPUT/OUTPUT

C.1.1 Programmed Input/Output

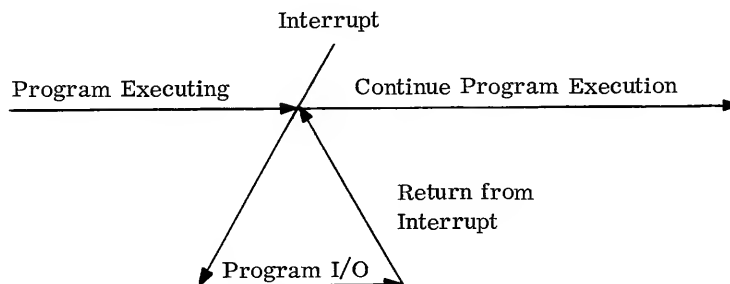
A programmed input/output operation is initiated by, and executed under control of the program:



The actual program steps required to enable programmed input/output depend on the design of the device controller.

C.1.2 Interrupt Input/Output

An interrupt input/output operation must be initiated by a peripheral device which transmits an interrupt to the CPU, causing the executing program to be interrupted for the duration of the input/output operation:



Interrupt input/output requires a definite and specific sequence of events, irrespective of what peripheral device is to be serviced; the sequence is as follows:

1. In order for an interrupt to be accepted by the CPU, the master interrupt enable flag (INEN) must be enabled (set to 1), and for the IMP-16L, the particular interrupt level corresponding to the device (IEN0 through IEN3) must be enabled also. If either flag is disabled (set to 0), interrupt signals from peripheral devices will be rejected. The master interrupt enable may be enabled by the instruction

LABEL: SFLG INEN

and disabled by the instruction

LABEL: PFLG INEN

The interrupt level flags may be manipulated via the following sequence of instructions:

PUSHF
PULL AC0

Code to modify flags

PUSH AC0
PULLF

2. Once an interrupt has been received and accepted by the CPU, the following steps occur automatically, and under control of the CPU.
 - a. The instruction currently being executed is completed, and the memory address of the next instruction is pushed onto the stack.
 - b. Interrupts are disabled (INEN is set to 0). Therefore no further interrupts will be accepted by the CPU until interrupts are re-enabled.
 - c. The instruction located at memory address X'0001 is executed.
3. The instruction at memory location X'0001 should be a jump to an interrupt service routine, which must perform a number of housekeeping tasks before the required input/output operation can proceed. Tasks, in order of normal execution follow.
 - a. Save the contents of accumulators and status register so that they can be restored just prior to returning from the interrupt. Accumulator and status register contents can be saved on the stack, but since the stack has just 16 words, more commonly a data area in memory is set aside for temporary data storage.
 - b. Determine the source of the interrupt. How this is done will depend on the design of the peripheral device controllers, but usually controllers are designed to follow the interrupt request signal by transmitting a data bit (or word) which identifies the source of the interrupt.
 - c. Once the interrupt has been identified, jump to the routine that services the identified device. This input/output service routine is written using programmed input/output as described in section 1. a.
4. Execute the selected device's input/output service routine.
5. Restore to the accumulator and status register contents that were saved in step 3a.
6. Return from the interrupt by executing an RTI instruction. This re-enables interrupts by setting flag INEN to 1 and pulls the return address (saved in the stack by step 2a) into the program counter, so execution continues at the program instruction following the interrupt.

C.1.3 Stack Full Interrupt

The Stack Full Condition (STKFUL) may optionally be connected to the interrupt enable; in this case, as soon as the bottom stack register (STK 15) is filled, the STKFUL condition is set and an interrupt is generated.

Processing following a "stack full" interrupt (level 0) requires essentially the same sequence of steps as outlined previously. The "stack full" interrupt must be identified by testing the "stack full" condition:

STKFUL = 8

LABEL: BOC STKFUL, HERE

If the "stack full" condition exists, execution will continue at HERE.

NOTE

Occurrence of a "stack full" interrupt may cause loss of data since it will result in the PC being pushed onto the stack. If a control panel interrupt immediately follows a "stack full" interrupt, the result may be a loss of two data words.

C.2 INPUT/OUTPUT SYSTEM ORGANIZATION

Depending on the intended application for the IMP-16, the type of input/output programming described in section C.1.1 may or may not be adequate.

In a dedicated application where the IMP-16 is used as a controller, or will rarely be subject to extensive reprogramming, it is efficient to incorporate input/output programming steps into the body of the program.

When the IMP-16 is to be constantly programmed, and particularly when peripheral devices are subject to change, it is more efficient to introduce the "logical unit" concept into input/output programming. Using this concept, programs are written to access peripheral devices functionally, rather than physically. For example,

Logical Unit 0 may be the operator interface device.
 Logical Unit 1 may be the bulk output device.
 Logical Unit 2 may be the bulk storage device.
 Logical Unit 3 may be the data entry device.
 Logical Units 4-7 may be data transmitting devices.

The operator interface device may be a Teletypewriter keyboard, or a (cathode ray tube) CRT terminal. The bulk output device may be a line printer or a Teletypewriter printer, or a paper tape punch.

The bulk storage device may be a magnetic disc, a magnetic tape, or a cassette unit.

The data entry device may be a Teletypewriter keyboard, the Teletypewriter paper tape reader or a high-speed paper tape reader.

The data transmitting devices may be analog-to-digital converters, intermediate magnetic storage devices, or specially wired external signal lines.

Input/output programming now has three parts:

- a. A generalized, logical unit oriented program to process requests for input/output.
- b. A set of device drivers that link the logical unit requested in "a" above with the required physical unit.
- c. Programs that actually enable the input/output operation.

C.2.1 Generalized Call to Input/Output

One subroutine will initiate all input/output operations with the exception of those operations that can only be initiated by an external interrupt. Let us call this subroutine IOS.

The execution of any input/output operation requested by a program will start with one common subroutine call

```

      LABEL      JSR      @IOS
                  .DWORD   LIST
  
```

where IOS provides, on the base page, the starting address of the input/output initiation subroutine, and LIST provides the memory address where the required input/output operation is defined. At LIST, the following information must be provided, using any convenient hexadecimal code:

1. The input/output operation to be performed should include:
 - a. Read
 - b. Write
 - c. Open (Initialize flags, counters or other conditions, if needed).

- d. Close (Provide device use termination processing, if needed).
 - e. Position to specified record and file.
 - f. Backspace (or forward space) a set number of records and/or files.
 - g. Return device status.
2. Input/output operation variables, including:
 - a. ASCII or binary for data transfers.
 - b. Echo or no echo for teletype.
 - c. Formatted or unformatted for printed output.
 3. Base address and length of memory buffer for read and write operations.
 4. Record and file number for position and backspace/forward space.

The IOS subroutine will interpret the information provided at LIST, then call the device driver for the physical unit corresponding to the requested logical unit. IOS will contain a physical unit assignment table to link physical units to logical units. For example, if there are eight logical units and six physical units, the table may take the form:

PUTBLE:	.WORD	X'0000	; LU 0 = PU 0
	.WORD	X'0100	; LU 1 = PU 0
	.WORD	X'0201	; LU 2 = PU 1
	.WORD	X'0302	; LU 3 = PU 2
	.WORD	X'0404	; LU 4 = PU 4
	.WORD	X'0503	; LU 5 = PU 3
	.WORD	X'0605	; LU 6 = PU 5
	.WORD	X'0705	; LU 7 = PU 5
	.WORD	X'0806	; LU 8 = PU 6

C. 2. 2 Device Drivers

The principal purpose of a device driver is to keep track of the status of peripheral devices during and between input/output calls. The device driver will maintain a device control block which is a data area dedicated to each peripheral device (one data area per device), where the following information is stored.

1. Busy/not busy. This serves two purposes.
 - a. To selectively disable/enable individual peripheral devices
 - b. To selectively disable other peripheral devices during certain phases of this device's operation
2. Record and file to which device was last positioned. The Open Call to IOS will reposition to this record and file, thus allowing reinitiation of discrete portions of input/output operations in the event of error conditions (bulk storage devices only).
3. Current record and file (bulk storage devices only).
4. Requested record and file (bulk storage devices only).
5. Selected parameters, coefficients, scale or conversion factors required or used by the device.
6. Condition of last operation: Successful, doubtful or error.

The device driver will now call the subroutine which executes the actual input/output operation.

Appendix D

PROGRAMMERS CHECKLIST

The following list of items is suggested for desk-checking a program prior to assembly.

1. Is the source program terminated by an `.END` Directive?
2. Is each label in the program terminated by a colon (`:`)?
3. Is each comment in the program preceded by a semi-colon (`;`)?
4. Is each string constant in the program set off on both ends by a prime (`'`)?
5. Is each Skip Instruction in the program followed by a single-word instruction?
6. Are all external symbols listed in `.GLOBL` Statements?
7. If using the Extended Instruction Set, has an `.EXTD` Directive been included?
8. Are all hexadecimal constants preceded by either `X'` or `0` (zero)?
9. For each `.IF` Directive in the program, is there a corresponding `.ENDIF`?
10. Are any global symbols defined by forward references? This is illegal.
11. Are any symbols defined by two-level forward references? This is illegal.

Appendix E

FOLD16 - IMP-16 FORTRAN OBJECT LOADER PROGRAM DESCRIPTION

FOLD16 is a FORTRAN subroutine that may be called by another FORTRAN program to input an IMP-16 Relocatable Load Module (RLM), convert it to core-image format, and place each portion (base sector, top sector, and so forth) in a vector representing the IMP-16 main memory in its proper location so that the program may be output to paper tape, cards, or other media for later loading into the IMP-16, or for producing PROMs of the program.

FOLD16 loads each element of the memory vector with one 16-bit word exactly as it is output from the IMP-16 assembler.

FOLD16 is written in ANSI FORTRAN to be executed on the system being employed for assembling IMP-16 programs. The only input/output performed by the subroutine consists of a series of reads of one or more IMP-16 relocatable load modules (output from the IMP-16 assembler).

E.1 GENERAL USAGE INFORMATION

One of the input parameters provided to FOLD16 by the user (see CALLING CONDITIONS) is a vector representing IMP-16 main memory and large enough to contain the modules that are to be loaded. A vector of "n" words will be assumed to represent IMP-16 main memory, addresses 0 through n-1. Since unless otherwise directed, the IMP-16 assembler generates absolute, base-sector, and top-sector code relative to address 0, the user must preallocate each portion of his code so that there will be no conflicts in memory. An example of how to do this follows.

```
.TITLE  MYPROG
.ASECT
.=25
  .
  .      code
  .
.BSECT
.=. +100
  .
  .      code
  .
.TSECT
.=. +1000
  .
  .      code
  .
.END  START
```

If more than one program is to be loaded, care must also be taken to avoid conflicts between programs. The user's attention is directed to the fact that even if he generates no actual base sector code, the IMP-16 assembler may, in resolving addresses, generate indirect address references through pointers which are actually allocated to base sector.

After the memory vector has been loaded as desired, the contents may be output to paper tape, for example, for leading into the IMP-16. The required format for an IMP-16 8-bit binary tape is described in the IMP-16 Utilities Manual (publication number 420003B) and the IMP-16L Utilities Manual (publication number 4200025A).

E.2 IMPLEMENTATION

The main subroutine entry point is FOLD16. The main subroutine calls one internal subroutine, UNPACK, which unpacks an IMP-16 word, 1 bit at a time, into 16 computer words.

The primary input data for FOLD16 is an IMP-16 RLM, the format of which is described in 6.4.

E.3 CALLING CONDITIONS

The calling sequence for FOLD16 is:

CALL FOLD16 (BIN,NEW,VECTOR)

where

BIN	is the FORTRAN logical unit number of the input/output device containing the input RLM.
NEW	The core-image vector into which the RLM specified by the user is loaded. This core-image vector is 4096 words long and represents IMP-16 memory, addresses 0 through 4095. When loaded, an element of this vector will contain an integer "n" in the range $0 \leq n \leq 65535$.
VECTOR	A 7-word vector in which FOLD16 returns loading statistics to the user.

<u>Word</u>	<u>Value</u>
1	Base sector initial address
2	Base sector final address
3	Top sector initial address
4	Top sector final address
5	Lowest memory address
6	Highest memory address
7	Module entry point

If more than one module is loaded, the first six elements of VECTOR will be updated as each module is completed. Element 7 will contain the entry point of the last RLM loaded.

Other tables used by FOLD16 are:

REC	A 16-word table into which FOLD16 reads an RLM record when processing it.
ARY	A 16-word table into which UNPACK unpacks computer words, one bit per element.

Care should be taken that the first set of records encountered by FOLD16 on the specified input/output device be a complete, well defined RLM (see 6.4).

Upon entry to FOLD16 for loading of the first RLM, the first six elements of VECTOR should be initialized to 65535_{10} , 0, 65535_{10} , 0, 65535_{10} , and 0. If a series of modules are to be loaded, the user may leave VECTOR unchanged until after the last RLM is loaded. He will then have cumulative statistics on the state of memory.

E.4 RETURNING CONDITIONS

Upon return to the caller, one RLM has been loaded from the unit specified in BIN, and the elements of VECTOR have been updated appropriately.

FOLD16 terminates its operation when it encounters an "END" record in the RLM. If an EOF condition occurs on the input device before an "END" record is found, the operating system terminates the program without returning control to the user program.

E.5 DESCRIPTION OF OPERATION

FOLD16 reads the RLM, record by record, and searches for "Data" records. When one is found, FOLD16 examines the record type and updates the corresponding set of loading statistics. It then strips out the data and stores it into the address specified in the record (and into locations following).

When the subroutine discovers an "END" record, it saves the entry point, updates the contents of VECTOR, and returns to its caller.

E.6 ENTRY NAME: UNPACK

E.6.1 Purpose

This subroutine unpacks a 16-bit integer into a 16-element array, one bit per element. The purpose is to facilitate bit manipulation in the FORTRAN calling program.

E.6.2 Calling Conditions

The calling sequence for UNPACK is

```
CALL UNPACK(VALU, ARY, BLOCK, LENG)
```

where

VALU is the word to be unpacked.

ARY is the 16-word array into which VALU is to be unpacked.

BLOCK is the starting bit number for unpacking.

LENG is the number of bits to be unpacked.

E.6.3 Returning Conditions

UNPACK returns to its caller when its task has been completed. The bit of VALU indicated by BLOCK* has been unpacked into ARY(1).

No error conditions are tested.

*Only the least significant 16 bits of VALU are considered, that is, the least significant bit is bit 16.

Appendix F

PROGRAM DIAGNOSTIC MESSAGES

F.1 INTRODUCTION

When a source program error is encountered by either the cross assembler or resident assembler, an appropriate error message, together with a pointer, is printed in the output (object listing). The pointer consists of an "@" or "?" character, depending upon the assembler program.

F.2 RESIDENT ASSEMBLER ERROR MESSAGES

The resident assembler only detects the first eight errors found in each statement. The error is diagnosed and marked in the listing, in the following line, by an error message (described below) and an @ character under the probable error field. An example of a resident assembler error detection is shown in figure F-1.

S	0003	0000	A	SH L	RO,1
ERROR UNDEFINED				@	

Figure F-1. Resident Assembler Error Detection, Listing Output

The following are the error messages:

- | | |
|-----------------------|---|
| 1. ERROR MISSING ARG. | This error indicates more arguments are required. |
| 2. ERROR VALUE | This error indicates value out of range or exceeds field size. |
| 3. ERROR ADDRESS | This error indicates address out of range. |
| 4. ERROR USAGE | This error indicates a number of possibilities including: <ul style="list-style-type: none"> a. An IF nesting error b. A local symbol in a global statement c. Symbol not previously defined which would affect location counter d. Wrong section e. Illegal expression, for example, external+5 |
| 5. ERROR SYNTAX | Indicates an illegal character or improper statement construction. |
| 6. ERROR EXCESS ARG. | Indicates the existence of unprocessed arguments. |
| 7. ERROR TBL OVERFLOW | <ul style="list-style-type: none"> a. If nesting level exceeds 10 b. Pointer table exceeds 50 pointers c. Number of local regions exceeds 64 d. Symbol table overflow |
| 8. ERROR UNDEFINED | Used to indicate either an undefined symbol or undefined instruction/directive. |
| 9. ERROR DUP. DEF. | Duplicate definition of the symbol. |
| 10. ERROR EXTD. INST. | Extended instruction illegally used although properly assembled. |

F.3 CROSS ASSEMBLER ERROR MESSAGES

Each error is diagnosed and marked with a "?" character in the following line of the output listing. The "?" is placed under the probable error field. The error is also marked on the listing with an asterisk (*) in column 1. Figure F-2 shows an example of cross assembler error detection.

	26 0000 0000 A M	=	C+1
*			?
	EXTERNAL CAN NOT APPEAR IN EXPRESSION		
	27 0000 0000 A D	=	B
*			?

Figure F-2. Cross Assembler Error Detection, Listing Output

1. ATTEMPT TO REDEFINE VALUE OF SYMBOL

Symbol, assigned a value in assignment statement, is already defined or a symbol changed value from pass 1 to pass 2.

2. CONDITIONAL ASSEMBLY ERROR

The conditional assembly directives do not balance. They must appear in sets of either .IF-.ENDIF or .IF-.ELSE-.ENDIF.

3. EXPRESSION VALUE EXCEEDS BOUNDS

The value of an expression is either illegal (for example, register value of 2 or 3 for SKAZ), or too large for field (for example, index register value of 4 or greater).

4. EXTERNAL CANNOT APPEAR IN EXPRESSION

A symbol defined in a separate assembly cannot appear in an expression.

5. EXTERNAL IN ASSIGNMENT

A symbol defined in a separate assembly cannot appear on the right side of an assignment statement.

6. GLOBAL UNDEFINED DURING PASS 1

An attempt was made to define a global symbol by a statement containing a forward reference.

7. ILLEGAL DIRECTIVE NAME

The directive flagged is not one recognized by the assembler.

8. ILLEGAL EXPRESSION

During evaluation of an expression, an illegal operator/operand combination was discovered.

9. ILLEGAL EXPRESSION MODE

Relocatable expression in register, flag, or jump-condition field, or value being assigned to location counter with mode different from location counter.

10. ILLEGAL FORM SYMBOL

The specified symbol is not recognizable as a legal operator or a defined .FORM symbol, or there was an error in the corresponding .FORM declaration.

11. ILLEGAL INDIRECT ADDRESSING

Indirect addressing is not legal in the specified instruction.

12. ILLEGAL INSTRUCTION

The instruction flagged is in the extended instruction set and a .EXTD directive has not been found in the program.

13. ILLEGAL SYNTAX

Instruction has incorrect structure (for example, operator not followed immediately by operand in an expression, .ASCII directive not followed by a 'string' , illegal character in an expression).

14. INTEGER EXCEEDS LIMITS

A decimal integer with a value less than -32768 or greater than 65535 or a hexadecimal value of more than four characters has been encountered.

15. LOCATION COUNTER OUTSIDE OF RANGE

Value of location counter in base sector exceeded FF_{16} or, in top or absolute sector, exceeded $FFFF_{16}$.

16. MULTIPLE DEFINITION

A symbol which appears in a .GLOBL statement or a .FORM statement or as a label is already defined.

17. OUT OF STORAGE

The maximum number of .FORM statements has been exceeded.

18. POINTER REGION OUT OF STORAGE

Assembler attempted to generate an indirect pointer in base sector and base sector was full, or maximum number of pointers has been exceeded.

19. SYMBOL XXXXXX UNDEFINED DUE TO SYMBOL TABLE OVERFLOW (Pass 1 Message)

Maximum number of symbols has been exceeded.

20. TOO MANY LOCAL DIRECTIVES

The maximum number of local directives has been exceeded.

21. TOO MANY OPERANDS

More operands appear in a .FORM call than appear in the corresponding declaration.

22. UNABLE TO GENERATE ADDRESS

Memory-reference instruction violates addressing limitations. See 5.4.

23. UNDEFINED SYMBOL

The symbol flagged is not defined in this program and does not appear in a .GLOBL statement.

F.4 OTHER ERROR CONDITIONS

If the assembly process is aborted by the operating system and a message is printed which indicates that an end-of-file condition was detected on the input file, the cause is probably omission of the .END directive at the end of the source program.

Appendix G

DIRECTIVE STATEMENTS

Table G-1. Index of Directive Statements

Statement	Operator Mnemonic	Operand Field
ASCII Directive	.ASCII	string [, string, ... string]
Program Section Directive	.ASECT	not used
Program Section Directive	.BSECT	not used
End Directive	.END	[address]
Extended Instruction Directive	.EXTD	not used
Form Directive	.FORM	symbol, exp [(exp) [, exp [(exp)]]]
Global Directive	.GLOBAL	symbol[, symbol, symbol... symbol]
Conditional	.IF	expression
	.ELSE	not used
	.ENDIF	not used
List Directive	.LIST	immediate
Local Directive	.LOCAL	not used
Page Directive	.PAGE	[string]
Space Directive	.SPACE	immediate
Title Directive	.TITLE	symbol[, string]
Program Section Directive	.TSECT	not used
Word Directive	.WORD	expression[, expression, ... exp]

Appendix H

INDEX OF INSTRUCTION STATEMENTS

Table H-1. Index of Basic Instruction Statements

Type	Instruction	Operation Mnemonic	First Field	Second Field	Address Class	Word Length
3	Add	ADD	reg	addr [(XR)]	1	1
9	Add Immediate Skip if Zero	AISZ	reg	immed		1
5	Logical AND	AND	accumulator	addr [(XR)]	1	1
8	Branch on Condition	BOC	immed4	spaddr		1
6	Complement and Add Immediate	CAI	reg	immed		1
9	Decrement, Skip if Zero	DSZ	addr [(XR)]		1	1
12	Halt	HALT				1
9	Increment, Skip if Zero	ISZ	addr [(XR)]		1	1
8	Jump	JMP	[@] addr [(XR)]		2	1
8	Jump to Subroutine	JSR	[@] addr [(XR)]		2	1
8	Jump to Sub- routine Implied	JSRI	addr		6	1
1	Load	LD	[@] reg [(XR)]	addr	2	1
6	Load Immediate	LI	reg	immed		1
	No Operation	NOP				1
5	Logical OR	OR	accumulator	addr [(XR)]	1	1
12	Pulse Flag	PFLG	immed3	[+immed]		1
6	Pull	PULL	reg			1
7	Pull Status Flag	PULLF				1
6	Push	PUSH	reg			1
7	Push Status Flag	PUSHF				1
6	Register Add	RADD	reg	reg		1
6	Register AND	RAND	reg	reg		1
6	Register Copy	RCPY	reg	reg		1
12	Register Input	RIN	+immed			1

Table H-1. Index of Basic Instruction Statements (Cont.)

Type	Instruction	Operation Mnemonic	First Field	Second Field	Address Class	Word Length
10	Rotate Left	ROL	reg	immed		1
10	Rotate Right	ROR	reg	immed		1
12	Register Output	ROUT	+immed			1
8	Return from Interrupt	RTI	[+immed]			1
8	Return from Subroutine	RTS	[+immed]			1
6	Register Exchange	RXCH	reg	reg		1
6	Register Exclusive OR	RXOR	reg	reg		1
12	Set Flag	SFLG	immed3	[+immed]		1
10	Shift Left	SHL	reg	immed		1
10	Shift Right	SHR	reg	immed		1
9	Skip If AND Is Zero	SKAZ	accumulator	addr [(XR)]	1	1
9	Skip Greater Than Zero	SKG	reg	addr [(XR)]	1	1
9	Skip Not Equal	SKNE	reg	addr [(XR)]	1	1
1	Store	ST	reg	[@]addr [(XR)]	2	1
3	Subtract	SUB	reg	addr [(XR)]	1	1
6	Exchange Reg. and Stack	XCHRS	reg			1

Table H-2. Index of Extended Instruction Statements

Type	Instruction	Operation Mnemonic	First Field	Second Field	Address Class	Word Length
7	Clear Bit	CLRBIT	immed4			1
7	Clear Status Flag	CLRST	immed4			1
7	Complement Bit	CMPBIT	immed4			1
4	Double-Precision Add	DADD	addr [(XR)]		3	2
3	Divide	DIV	addr [(XR)]		3	2
4	Double-Precision Subtract	DSUB	addr [(XR)]		3	2
11	Interrupt Scan	ISCAN				1
8	Jump to Level 0 Interrupt Indirect	JINT	immed4		5	1
8	Jump through Pointer	JMPP	immed4		7	1
8	Jump to Subroutine Pointer	JSRP	+immed		7	1
2	Load Byte	LDB	addr [(XR)]		4	2
2	Load Left Byte	LLB	addr [(XR)]		4	2
2	Load Right Byte	LRB	addr [(XR)]		4	2
3	Multiply	MPY	addr [(XR)]		3	2
7	Set Bit	SETBIT	immed4			1
7	Set Status Flag	SETST	immed4			1
9	Skip if Bit True	SKBIT	immed4			1
9	Skip if Status Flag True	SKSTF	immed4			1
2	Store Left Byte	SLB	addr [(XR)]		4	2
2	Store Right Byte	SRB	addr [(XR)]		4	2
2	Store Byte	STB	addr [(XR)]		4	2

Table H-3. Definitions of Field Designators

Field Designator	Definition
@	indirect addressing
accumulator	register field (value = 0 or 1)
addr	address field
immed	immediate operand
immed3	positive 3-bit immediate operand
immed4	positive 4-bit immediate operand
+immed	positive immediate operand
reg	register field
spaddr	special address field
(xr)	register field (value = 2 or 3)

Appendix I

CONVERSION TABLES

TABLE I-1

POSITIVE POWERS OF TWO

n	2^n	n	2^n
1	2	51	22517 99813 68524 8
2	4	52	45035 99627 37049 6
3	8	53	90071 99254 74099 2
4	16	54	18014 39850 94819 84
5	32	55	36028 79701 89639 68
6	64	56	72057 59403 79279 36
7	128	57	14411 51880 75855 872
8	256	58	28823 03761 51711 744
9	512	59	57646 07523 03423 488
10	1024	60	11529 21504 60684 6976
11	2048	61	23058 43009 21369 3952
12	4096	62	46116 86018 42738 7904
13	8192	63	92233 72036 85477 5808
14	16384	64	18446 74407 37095 51616
15	32768	65	36893 48814 74191 03232
16	65536	66	73786 97629 48382 06464
17	13107 2	67	14757 39525 89676 41292 8
18	26214 4	68	29514 79051 79352 82585 6
19	52428 8	69	59029 58103 58705 65171 2
20	10485 76	70	11805 91620 71741 13034 24
21	20971 52	71	23611 83241 43482 26068 48
22	41943 04	72	47223 66482 86964 52136 96
23	83886 08	73	94447 32965 73929 04273 92
24	16777 216	74	18889 46593 14785 80854 784
25	33554 432	75	37778 93186 29571 61709 568
26	67108 864	76	75557 86372 59143 23419 136
27	13421 7728	77	15111 57274 51828 64683 8272
28	26843 5456	78	30223 14549 03657 29367 6544
29	53687 0912	79	60446 29098 07314 58735 3088
30	10737 41824	80	12089 25819 61462 91747 06176
31	21474 83648	81	24178 51639 22925 83494 12352
32	42949 67296	82	48357 03278 45851 66988 24704
33	85899 34592	83	96714 06556 91703 33976 49408
34	17179 86918 4	84	19342 81311 38340 66795 29881 6
35	34359 73836 8	85	38685 62622 76681 33590 59763 2
36	68719 47673 6	86	77371 25245 53362 67181 19526 4
37	13743 89534 72	87	15474 25049 10672 53436 23905 28
38	27487 79069 44	88	30948 50098 21345 06872 47810 56
39	54975 58138 88	89	61897 00196 42690 13744 95621 12
40	10995 11627 776	90	12379 40039 28538 02748 99124 224
41	21990 23255 552	91	24758 80078 57076 05497 98248 448
42	43980 46511 104	92	49517 60157 14152 10995 96496 896
43	87960 93022 208	93	99035 20314 28304 21991 92993 792
44	17592 18604 4416	94	19807 04062 85660 84398 38598 7584
45	35184 37208 8832	95	39614 08125 71321 68796 77197 5168
46	70368 74417 7664	96	79228 16251 42643 37593 54395 0336
47	14073 74883 55328	97	15845 63250 28528 67518 70879 00672
48	28147 49767 10656	98	31691 26500 57057 35037 41758 01344
49	56294 99534 21312	99	63382 53001 14114 70074 83516 02688
50	11258 99906 84262 4	100	12676 50600 22822 94014 96703 20537 6
		101	25353 01200 45645 88029 93406 41075 2

I-2

NEGATIVE POWERS OF TWO

n	2 ⁻ⁿ								
0	1.0								
1	0.5								
2	0.25								
3	0.125								
4	0.0625								
5	0.03125								
6	0.015625	5							
7	0.0078125	25							
8	0.00390625	625							
9	0.001953125	3125							
10	0.0009765625	65625							
11	0.00048828125	5							
12	0.000244140625	25							
13	0.0001220703125	125							
14	0.00006103515625	5625							
15	0.000030517578125	78125							
16	0.0000152587890625	5							
17	0.00000762939453125	25							
18	0.000003814697265625	625							
19	0.0000019073486328125	8125							
20	0.00000095367431640625	40625							
21	0.000000476837158203125	5							
22	0.0000002384185791015625	25							
23	0.00000011920928955078125	125							
24	0.000000059604644775390625	625							
25	0.0000000298023223876953125	53125							
26	0.00000001490116119384765625	5							
27	0.000000007450580596923828125	25							
28	0.0000000037252902984619140625	625							
29	0.00000000186264514923095703125	3125							
30	0.000000000931322574615478515625	5							
31	0.0000000004656612873077392578125	25							
32	0.00000000023283064365386962890625	125							
33	0.000000000116415321826934814453125	625							
34	0.0000000000582076609134674072265625	5625							
35	0.00000000002910383045673370361328125	28125							
36	0.000000000014551915228366851806640625	5							
37	0.0000000000072759576141834259033203125	25							
38	0.00000000000363797880709171295166015625	625							
39	0.000000000001818989403545856475830078125	8125							
40	0.0000000000009094947017729282379150390625	90625							
41	0.00000000000045474735088646411895751953125	5							
42	0.000000000000227373675443232059478759765625	25							
43	0.0000000000001136868377216160297393798828125	125							
44	0.0000000000000568434188608081486968994140625	625							
45	0.000000000000028421709430404007434844970703125	5							
46	0.0000000000000142108547152020037174224853515625	25							
47	0.00000000000000710542735760100185871124267578125	625							
48	0.000000000000003552713678800500929355621337890625	3125							
49	0.0000000000000017763568394002504646778106689453125	625							
50	0.00000000000000088817841970012523233890533447265625	625							

TABLE I-3

HEXADECIMAL AND DECIMAL INTEGER CONVERSION TABLE

8		7		6		5		4		3		2		1	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
8		7		6		5		4		3		2		1	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

To convert integer numbers greater than the capacity of table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

$$\begin{array}{r}
 D = 13 \\
 \times 16 \\
 \hline
 208 \\
 3 = +3 \\
 \hline
 211 \\
 \times 16 \\
 \hline
 3376 \\
 4 = +4 \\
 \hline
 3380
 \end{array}$$

EXAMPLE	
Conversion of Hexadecimal Value	D34
1. D	3328
2. 3	48
3. 4	4
4. Decimal	3380

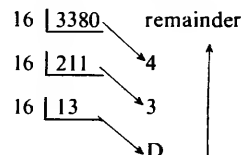
TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
(b) Record the hexadecimal of the column containing the selected number.
(c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order.

Example: $3380_{10} = D34_{16}$

EXAMPLE

Conversion of Decimal Value	3380
1. D	<u>-3328</u> 52
2. 3	<u>-48</u> 4
3. 4	<u>-4</u> 0
4. Hexadecimal	D34

TABLE I-4

HEXADECIMAL AND DECIMAL FRACTION CONVERSION TABLE

1		2		3				4			
Hex	Decimal	Hex	Decimal	Hex	Decimal			Hex	Decimal Equivalent		
.0	.0000	.00	.0000 0000	.000	.0000	0000	0000	.0000	.0000	0000	0000
.1	.0625	.01	.0039 0625	.001	.0002	4414	0625	.0001	.0000	1525	8789 0625
.2	.1250	.02	.0078 1250	.002	.0004	8828	1250	.0002	.0000	3051	7578 1250
.3	.1875	.03	.0117 1875	.003	.0007	3242	1875	.0003	.0000	4577	6367 1875
.4	.2500	.04	.0156 2500	.004	.0009	7656	2500	.0004	.0000	6103	5156 2500
.5	.3125	.05	.0195 3125	.005	.0012	2070	3125	.0005	.0000	7629	3945 3125
.6	.3750	.06	.0234 3750	.006	.0014	6484	3750	.0006	.0000	9155	2734 3750
.7	.4375	.07	.0273 4375	.007	.0017	0898	4375	.0007	.0001	0681	1523 4375
.8	.5000	.08	.0312 5000	.008	.0019	5312	5000	.0008	.0001	2207	0312 5000
.9	.5625	.09	.0351 5625	.009	.0021	9726	5625	.0009	.0001	3732	9101 5625
.A	.6250	.0A	.0390 6250	.00A	.0024	4140	6250	.000A	.0001	5258	7890 6250
.B	.6875	.0B	.0429 6875	.00B	.0026	8554	6875	.000B	.0001	6784	6679 6875
.C	.7500	.0C	.0468 7500	.00C	.0029	2968	7500	.000C	.0001	8310	5468 7500
.D	.8125	.0D	.0507 8125	.00D	.0031	7382	8125	.000D	.0001	9836	4257 8125
.E	.8750	.0E	.0546 8750	.00E	.0034	1796	8750	.000E	.0002	1362	3046 8750
.F	.9375	.0F	.0585 9375	.00F	.0036	6210	9375	.000F	.0002	2888	1835 9375
1		2		3				4			

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250
 Find .0B in position 2 .0429 6875
 Find .00C in position 3 .0029 2968 7500
 .ABC Hex is equal to .6708 9843 7500

TABLE I-5

INTEGER CONVERSION TABLE**POWERS OF 16 TABLE**

Example: $268,435,456_{10} = (2.68435456 \times 10^8)_{10} = 1000\ 0000_{16} = (10^7)_{16}$

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10 = A
17 592 186 044 416	11 = B
281 474 976 710 656	12 = C
4 503 599 627 370 496	13 = D
72 057 594 037 927 936	14 = E
1 152 921 504 606 846 976	15 = F
Decimal Values	

NEGATIVE HEXADECIMAL NUMBERS

The IMP-16 maintains negative numbers in twos-complement form. To convert a number in hexadecimal notation to its twos-complement equivalent, subtract the number from 2^n expressed in hexadecimal form. The number "n" is the number of binary bits in the computer word. For example, if the computer uses a 16-bit word, the number "n" is equal to 16. Thus, the negative of 1245₁₆ is derived as follows:

1 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
<u>- 1 2 4 5</u>	<u>- 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1</u>
E D B B	1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1

Note that a hexadecimal number will be negative in the IMP-16 computer if the left most digit is 8, 9, A, B, C, D, E, or F. Thus, FACE is equal to 1111 1010 1100 1110; the twos complement is:

1 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
<u>- F A C E</u>	<u>1 1 1 1 1 0 1 0 1 1 0 0 1 1 1 0</u>
+ 5 3 2	- 0 0 0 0 0 1 0 1 0 0 1 1 0 0 1 0

Appendix J

REFERENCES

1. IMP-16L Users Manual (IMP-16L/928)
Describes the IMP-16L microcomputer equipment and instruction set.
2. IMP-16C Application Manual (IMP-16/925)
Describes the IMP-16C microprocessor equipment and instruction set.
3. IMP-16 Utilities Reference Manual (IMP-16/925)
Describes (1) the debugging and loading programs applicable to the IMP-16L and IMP-16P microcomputer, and (2) operating procedures for these programs based on use of the IMP-16L and IMP-16P control panel.
4. Tymshare Users Manual (IMP-00S/118Y)
Describes the use of the TYMSHARE (nation-wide) timesharing system for implementing the IMP-16 Assembler and its related programs.
5. IMP-16P Users Manual (IMP-16P/937)
Describes the IMP-16P microcomputer system.



National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
(408) 732-5000
TWX: 910-339-9240

National Semiconductor Electronics SDNBHD
Batu Berendam
Free Trade Zone
Malacca, Malaysia
Telephone: 5171
Telex: NSELECT 519 MALACCA (c/o Kuala Lumpur)

National Semiconductor GmbH
D 808 Fuerstenfeldbruck
Industriestrasse 10
West Germany
Telephone: (08141) 1371
Telex: 27649

National Semiconductor (UK) Ltd.
Larkfield Industrial Estates
Greenock, Scotland
Telephone: (0475) 33251
Telex: 778 632

National Semiconductor (Pte.) Ltd.
No. 1100 Lower Delta Rd.
Singapore 3
Telephone: 630011
Telex: 21402

REGIONAL AND DISTRICT SALES OFFICES

ALABAMA

DIXIE DISTRICT OFFICE
3322 Memorial Pkwy, S.W. #67
Huntsville, Alabama 35802
(205) 881-0622
TWX: 810-726-2207

ARIZONA

*ROCKY MOUNTAIN REGIONAL OFFICE
3313 North 68th Street, No. 114
Scottsdale, Arizona 85251
(602) 945-8473

CALIFORNIA

*NORTH-WEST REGIONAL OFFICE
2680 Bayshore Frontage Road, Suite 112
Mountain View, California 94043
(415) 961-4740
TWX: 910-379-6432

*SOUTH-WEST REGIONAL OFFICE
Valley Freeway Center Building
15300 Ventura Boulevard, Suite 305
Sherman Oaks, California 91403
(213) 783-8272
TWX: 910-495-1773

DISTRICT SALES OFFICE

17452 Irvine Boulevard, Suite M
Tustin, California 92680
(714) 832-8113
TWX: 910-595-1523

CONNECTICUT

AREA OFFICE
Commerce Park
Danbury, Connecticut 06810
(203) 744-2350

*DISTRICT SALES OFFICE

25 Sylvan Road South
Westport, Connecticut 06880
(203) 226-6833

FLORIDA

*AREA SALES OFFICE
2721 South Bayshore Drive, Suite 121
Miami, Florida 33133
(305) 446-8309
TWX: 810-848-9725

CARIBBEAN REGIONAL SALES OFFICE

P.O. Box 6335
Clearwater, Florida 33518
(813) 441-3504

ILLINOIS

NORTH-CENTRAL REGIONAL OFFICE
800 E. Northwest Highway, Suite 1060
Palatine, Illinois 60067
(312) 693-2660
TWX: 910-693-4805

INDIANA

DISTRICT SALES OFFICE
P.O. Box 40073
Indianapolis, Indiana 46240
(317) 255-5822

KANSAS

DISTRICT SALES OFFICE
13201 West 82nd Street
Lenexa, Kansas 66215
(816) 358-8102

MARYLAND

CAPITAL REGIONAL SALES OFFICE
300 Hospital Drive, No. 232
Glen Burnie, Maryland 21061
(301) 760-5220
TWX: 710-861-0519

MASSACHUSETTS

*NORTH-EAST REGIONAL OFFICE
No. 3 New England, Exec. Office Park
Burlington, Massachusetts 01803
(617) 273-1350
TWX: 710-332-0166

MICHIGAN

*DISTRICT SALES OFFICE
23629 Liberty Street
Farmington, Michigan 48024
(313) 477-0400

MINNESOTA

DISTRICT SALES OFFICE
9701 Penn Avenue S., Suite 109
Minneapolis, Minnesota 55431
(612) 888-4666
TWX: 910-576-3415

NEW JERSEY/NEW YORK CITY

MID-ATLANTIC REGIONAL OFFICE
301 Sylvan Avenue
Englewood Cliffs, New Jersey 07632
(201) 871-4410
TWX: 710-991-9734

NEW YORK (UPSTATE)

CAN-AM REGIONAL SALES OFFICE
104 Pickard Drive
Syracuse, New York 13211
(315) 455-5858

OHIO/PENNSYLVANIA/ W. VIRGINIA/KENTUCKY

EAST-CENTRAL REGIONAL OFFICE
Financial South Building
5335 Far Hills, Suite 214
Dayton, Ohio 45429
(513) 434-0097

TEXAS

*SOUTH-CENTRAL REGIONAL OFFICE
5925 Forest Lane, Suite 205
Dallas, Texas 75230
(214) 233-6801
TWX: 910-860-5091

WASHINGTON

DISTRICT OFFICE
300 120th Avenue N.E.
Building 2, Suite 205
Bellevue, Washington 98005
(206) 454-4600

INTERNATIONAL SALES OFFICES

AUSTRALIA

*NATIONAL SEMICONDUCTOR
ELECTRONICS PTY, LTD.
Cnr. Stud Road & Mountain Highway
Bayswater, Victoria 3153
Australia
Telephone: 729-0733
Telex: 32096

CANADA

*NATIONAL SEMICONDUCTOR CORP.
1111 Finch Avenue West
Downsview, Ontario, Canada
(416) 635-9880
TWX: 610-492-2510

DENMARK

NATIONAL SEMICONDUCTOR
SCANDINAVIA
Vordingborggade 22
2100 Copenhagen
Denmark
Telephone: (01) 92-OBRO-5610
Telex: DK 6827 MAGNA

ENGLAND

NATIONAL SEMICONDUCTOR (UK) LTD.
The Precinct
Broxbourne, Hertfordshire
England
Telephone: 69571
Telex: 267-204

FRANCE

NATIONAL SEMICONDUCTOR
FRANCE S.A.R.L.
28, Rue de la Redoute
92260-Fontenay-Aux-Roses
Telephone: 660-81-40
TWX: NSF 25956F

HONG KONG

*NATIONAL SEMICONDUCTOR
HONG KONG LTD.
9 Lai Yip Street
Kwun Tung, Kowloon
Hong Kong
Telephone: 3-458888
Telex: HX3866

JAPAN

*NATIONAL SEMICONDUCTOR JAPAN
Nakazawa Building
1-19 Yotsuya, Shinjuku-Ku
Tokyo, Japan 160
Telephone: 03-359-4571
Telex: J 28592

SWEDEN

NATIONAL SEMICONDUCTOR SWEDEN
Sikvagen 17
13500 Tyreso
Stockholm
Sweden
Telephone: (08) 712-04-80

WEST GERMANY

*NATIONAL SEMICONDUCTOR GMBH
8000 Munchen 81
Cosimstrasse 4
Telephone: (0811) 915-027